中国电子教育学会高教分会推荐
高等学校应用型本科“十三五”规划教材

机 电 MECHATRONICS
计算机 COMPUTER
电 子 ELECTRONICS

- 突破传统的侧重技术细节的形式，以“项目驱动、任务导向”的方式组织内容。
- 配有案例源代码与课件，方便使用。

Java EE 框架技术

(SpringMVC+Spring+MyBatis)

主　编　陈永政　张正龙
副主编　谢东亮　张红实　李冀明

西安电子科技大学出版社
http://www.xduph.com

站在巨人的肩上

Standing on Shoulders of Giants

中国电子教育学会高教分会推荐

高等学校应用型本科“十三五”规划教材

Java EE框架技术

(SpringMVC+Spring+MyBatis)

主　编　陈永政　张正龙

副主编　谢东亮　张红实　李冀明

西安电子科技大学出版社

内 容 简 介

本书对当前企业使用较多的、流行的三大技术框架 SpringMVC、Spring 和 MyBatis 的基本知识和使用方法进行了详细的讲解。全书共分为 7 章。第一章主要介绍 MyBatis 开发入门知识；第二章主要介绍 MyBatis 配置选项；第三章主要介绍 MyBatis 映射器(Mapper)；第四章主要介绍 Spring 核心技术；第五章主要介绍 SpringMVC；第六章主要介绍 SpringMVC、Spring、MyBatis 三个框架的集成；第七章主要是项目实战部分。本书在讲解知识点的同时还提供了丰富的案例，每章节末均给出一定量的练习题，以帮助学生巩固学习效果，加深对相关知识点的理解。

本书可作为高等院校计算机相关专业软件工程类课程的教材，也可作为相关工程技术人员的参考用书。

图书在版编目(CIP)数据

Java EE 框架技术：SpringMVC+Spring+MyBatis/陈永政，张正龙主编.
—西安：西安电子科技大学出版社，2017.8
ISBN 978-7-5606-4589-6

Ⅰ. ① J… Ⅱ. ① 陈… ② 张… Ⅲ. ① JAVA 语言—程序设计 Ⅳ. ① TP312

中国版本图书馆 CIP 数据核字(2017)第 171756 号

策　　划　李惠萍
责任编辑　黄菡　阎彬
出版发行　西安电子科技大学出版社(西安市太白南路 2 号)
电　　话　(029)88242885　88201467　　邮　　编　710071
网　　址　www.xduph.com　　电子邮箱　xdupfxb001@163.com
经　　销　新华书店
印刷单位　陕西利达印务有限责任公司
版　　次　2017 年 8 月第 1 版　　2017 年 8 月第 1 次印刷
开　　本　787 毫米×1092 毫米　1/16　印　张　14
字　　数　325 千字
印　　数　1～3000 册
定　　价　26.00 元

ISBN 978-7-5606-4589-6/TP

XDUP 4881001-1

前　言

本书介绍的 SpringMVC、Spring、MyBatis 开源框架开发技术，是基于 Java EE 的 Web 应用程序，用于开发企业级 Web 应用的软件框架。Java EE 通过提供中间层集成框架来满足各种应用需求。Java EE 架构具有高可用性、高可靠性、高扩展性，并且成本低，是企业构建 Web 应用平台的首选。而 Java EE 架构通常选用 SpringMVC+Spring+MyBatis 框架作为其基础开发框架。通过对三个框架的合理整合，不仅可以大幅度提高系统的开发效率，而且能提高系统的稳定性、健壮性与安全性。

本书介绍了 Java EE 开发使用的三大开发框架 SpringMVC、Spring、MyBatis 及其整合使用。本书内容共分为七章。第一章为 MyBatis 开发入门知识，讨论了 MyBatis 的优势，使用 MyBaits 访问数据库的优点，并创建了第一个 MyBatis 项目，带领读者进入 MyBatis。第二章为 MyBatis 配置选项，使用基于 XML 配置和基于 Java API 配置的方式引导 MyBatis。第三章为映射器配置，是本书重点，讨论了怎样使用映射器配置文件书写 SQL 映射语句，如何配置简单的语句、一对一以及一对多关系的语句，以及怎样使用 resultMap 进行结果集映射；还讨论了动态 SQL 的书写方法及使用注解书写 SQL 映射语句，最后介绍如何使用 MyBatis Generator 自动创建实体类、接口、配置文件代码。第四章为 Spring 核心技术，讨论了 Spring 的核心知识，包括 Spring IoC 和 Spring AOP 技术。第五章为 SpringMVC 框架技术，包括 SpringMVC 概述、创建第一个 SpringMVC 程序、SpringMVC RequestMapping 的基本设置、SpringMVC 前后台数据交互、SpringMVC 文件上传下载，以及 SpringMVC 常用注解。第六章为 SpringMVC、Spring、MyBatis 的集成部分，介绍了三个框架的集成步骤。第七章为项目实战部分，主要以云服务器租赁后台管理系统为导向，介绍了综合应用 SpringMVC+Spring+MyBatis 框架实现一个项目的技术与过程。

本书突破传统的侧重 Java EE 技术细节介绍的形式，以“项目驱动、任务导向”的方式进行内容组织。首先以项目案例的实现为先导，让读者了解某项技术的应用，引起读者对这些技术实现的兴趣，激起其探索该技术实现原理与理论知识的愿望。然后通过有目的的学习，让读者掌握书中介绍的知识点及实现技术。本书介绍的相关技术具有连贯性。

本书适合作为高等院校计算机相关专业软件工程类课程的教材，也适合作为相关工程技术人员的参考用书。本书配有一系列案例源代码，这些案例代码均经过调试可以运行。书中介绍了这些案例的实现过程，读者可以按照书中介绍的案例实现步骤自行实现，并可借助这些案例引导，逐步掌握使用 SpringMVC、Spring、MyBatis 框架进行综合应用软件项目的开发。本书相关源码下载地址：https://github.com/bay-test/ssmbooksource。

本书由陈永政和张正龙担任主编，陈永政主要承担了第一章、第二章、第三章、第五章、第六章的编写；张正龙主要承担了第四章、第七章的编写；谢东亮、张红实、李冀明承担了部分章节的编写，并提出了大量有益的建议。重庆知人者科技有限公司的沈国仿工程师参与了本书教学案例的设计及教学内容的设计，在此一并表示感谢。

由于时间仓促及编者水平有限，书中难免存在疏漏和不足之处，恳请同行专家和读者给予批评与指正。

编者邮箱：610919606@qq.com

编　者

2017 年 5 月

目　录

第一章　MyBatis 开发入门

欢迎您来到 MyBatis 的世界，MyBatis 是一个支持普通 SQL 查询、存储过程和高级映射的优秀持久层框架。MyBatis 消除了几乎所有的 JDBC 代码和参数的手工设置以及对结果集的检索封装。MyBatis 可以使用简单的 XML 或注解来配置和进行原始映射，将接口和 Java 的 POJOs(Plain Old Java Objects，普通的 Java 对象)映射成数据库中的记录。

本章知识要点

- MyBatis 简介
- MyBatis 的优势
- 认识第一个 MyBatis 程序
- MyBatis 日志

1.1　MyBatis 简介

MyBatis 本是 apache 的一个开源项目 iBatis，2010 年这个项目由 apache software foundation 迁移到了 google code，并且改名为 MyBatis。2013 年 11 月迁移到 Github。

iBatis 一词来源于“internet”和“abatis”的组合，是一个基于 Java 的持久层框架。iBatis 提供的持久层框架包括 sql Maps 和 Data Access Objects(DAO)。

MyBatis 的功能架构分为三层，如图 1-1 所示。

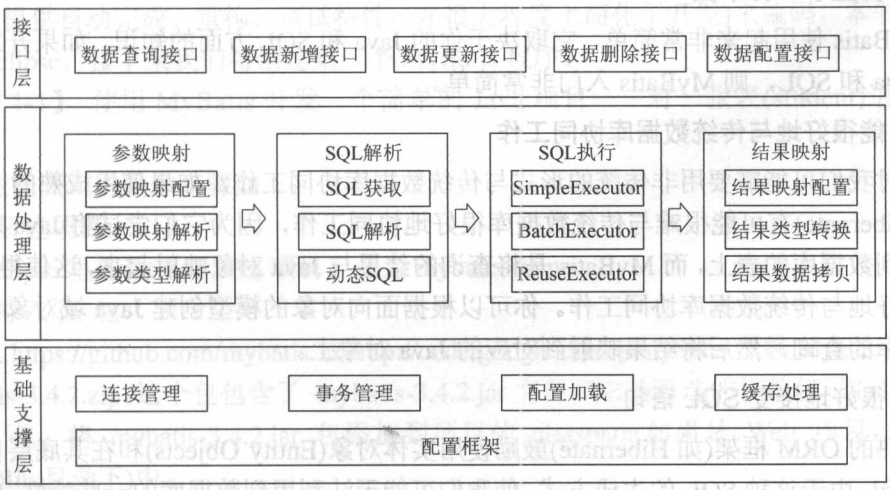

图 1-1　MyBatis 的功能架构

(1) 接口层：提供给外部使用的接口 API，开发人员通过这些本地 API 来操纵数据库。接口层一接收到调用请求就会调用数据处理层来完成具体的数据处理。

(2) 数据处理层：负责具体的参数映射、SQL 解析、SQL 执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作。

(3) 基础支撑层：负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这些共用的功能，被抽取出来作为最基础的组件，为上层的数据处理层提供最基础的支撑。

MyBatis 应用程序主要使用 sqlSessionFactory 实例，一个 sqlSessionFactory 实例可以通过 sqlSessionFactoryBuilder 获得。sqlSessionFactoryBuilder 可以从一个 XML 配置文件或者一个预定义的配置类的实例获得。

1.2 MyBatis 的优势

1. 消除了大量的 JDBC 冗余代码

Java 通过 Java 数据库连接(Java Data Base Connectivity，JDBC)API 来操作关系型数据库，但是 JDBC 是一个非常底层的 API，我们需要书写大量的代码来完成对数据库的操作。MyBatis 通过简单的方式实现了和 JDBC 相同的功能，如准备需要被执行的 sql Statement 对象并且将 Java 对象作为输入数据传递给 statement 对象的任务，开发人员可以专注于真正重要的方面。另外，MyBatis 使将输入的 Java 对象中的属性设置成查询参数和从 SQL 结果集上生成 Java 对象这两个过程自动化。MyBatis 还提供了其他的一些特性使持久化逻辑的实现变得简单：

(1) 它支持复杂的 SQL 结果集数据映射到嵌套对象图结构；

(2) 它支持一对一和一对多的结果集和 Java 对象的映射；

(3) 它支持根据输入的数据构建动态的 SQL 语句。

2. 易上手和易掌握

MyBatis 使用起来非常简单，它取决于你的 Java 和 SQL 方面的知识。如果开发人员很熟悉 Java 和 SQL，则 MyBatis 入门非常简单。

3. 能很好地与传统数据库协同工作

有时我们可能需要用非传统的形式与传统数据库协同工作，如果使用成熟的 ORM 框架(如 Hibernate)有可能很难与传统数据库很好地协同工作，因为它们尝试将 Java 对象静态地映射到数据库的表上，而 MyBatis 是将查询的结果与 Java 对象映射起来，这使得 MyBatis 可以很好地与传统数据库协同工作。你可以根据面向对象的模型创建 Java 域对象，执行传统数据库的查询，然后将结果映射到对应的 Java 对象上。

4. 很好地接受 SQL 语句

成熟的 ORM 框架(如 Hibernate)鼓励使用实体对象(Entity Objects)和在其底层自动产生 SQL 语句。由于这种 SQL 的生成方式，使我们可能无法利用到数据库的一些特性。Hibernate 允许执行本地 SQL，但是这样会打破持久层和数据库独立的原则。

MyBatis 框架接受 SQL 语句，而不是将其对开发人员隐藏起来。由于 MyBatis 不会产生任何 SQL 语句，所以开发人员要自己准备 SQL 语句，这样就可以充分利用数据库特有的特性并且可以准备自定义的查询。另外，MyBatis 对存储过程也提供支持。

5. 提供与第三方缓存类库的集成支持

MyBatis 有内建的 sqlSession 级别的缓存机制，可缓存 select 语句的查询结果。除此之外，MyBatis 还提供与多种第三方缓存类库的集成支持，如 EHCache、OSCache、Hazelcast 等。

6. 引入了更好的性能

性能是关乎软件应用成功与否的关键因素之一。为了获得更好的性能，需要考虑很多方面，而对很多应用而言，数据持久化层是整个系统性能的关键。

MyBatis 支持数据库连接池，消除了为每一个请求创建一个数据库连接的开销。

MyBatis 提供内建的缓存机制，在 sqlSession 级别提供对 SQL 查询结果的缓存。即：如果你调用了相同的 select 查询，MyBatis 会将放在缓存的结果返回，而不会再去查询数据库。

MyBatis 框架并没有大量地使用代理机制，因此对于其他的过度使用代理的 ORM 框架而言，MyBatis 可以获得更好的性能。

1.3 认识第一个 MyBatis 程序

在本节开始之前假设你的系统上已经安装了 JDK1.6+(下载网站：http://www.java. com)和 MySQL5(下载网站：http://www.mysql.com)。JDK 和 MySQL 的安装过程不在本书的叙述范围之内，所以这里不再阐述。

目前 MyBatis 的最新版本是 MyBatis 3.4.2，本书将使用 MyBatis 3.4.2 版本。

本书并不限定你使用什么类型的 IDE(如 Eclipse、NetBeans IDE 或者 IntelliJ IDEA IDE，它们通过提供自动完成、重构、调试特性，在很大程度上简化了开发)来编码。本书采用的 IDE 是 Eclipse。接下来我们将创建第一个 MyBatis 程序。

【例 1-1】 使用 MyBatis 开发一个简单的 Java 项目——对一张表(student)进行数据查询。

➢ 数据查询的具体步骤如下：

(1) 新建项目。

在 eclipse 里创建一个 Java 项目，名为 mybatis-demo。

(2) 导入 jar 包。

进入 https://github.com/mybatis 或 http://code.google.com/p/mybatis 下载 MyBatis 的发布包 mybatis-3.4.2.zip。这个包包含了 mybatis-3.4.2.jar 文件和它的可选的依赖包，如 slf4j/log4j 日志 jar 包，将 mybatis-3.4.2.jar 包添加到项目的 classpath(如果是 Web 项目，可放在 Web-INF\lib 目录下)中。

进入 http://www.mysql.com/products/connector/ 下载 mysql 的 Java 驱动包 mysql-

connector-java-5.1.22.jar 并将其添加到项目 classpath 中。

如果项目正在使用 maven，那么配置这些 jar 包的依赖就变得简单多了。在 pom.xml 中添加以下依赖即可：

```
<dependencies>
<!-- MyBatis 依赖 -->
<dependency>
<groupId>org.mybatis</groupId>
<artifactId>mybatis</artifactId>
<version>3.4.2</version>
</dependency>
<!-- mysql 的 java 驱动包依赖 -->
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>5.1.22</version>
</dependency>
</dependencies>
```

更多 maven 的相关信息请参照 http://maven.apache.org/ 学习。

(3) 在 MySQL 里新建表并插入样本数据。

使用 SQL 脚本生成 student 表，并且插入三条样本数据，脚本如下：

```
-- 建表
CREATE TABLE 'student' (
  'stuId' int(10) NOT NULL AUTO_INCREMENT,
  'stuName' varchar(50) DEFAULT NULL,
  PRIMARY KEY ('stuId')
) ENGINE = InnoDB AUTO_INCREMENT = 1 DEFAULT CHARSET = utf8;
-- 插入三条样本数据
INSERT INTO 'student' VALUES ('1', 'zhangsan');
INSERT INTO 'student' VALUES ('2', 'lisi');
INSERT INTO 'student' VALUES ('3', 'wangwu');
```

(4) 从 XML 映射 SQL 语句。

在 classpath(如果是 Web 项目，classpath 在 WEB-INF\classes 目录下)下 com.test.mapper 包里新建 StudentMapper.xml 文件，将 SQL 信息进行映射，其配置内容如下：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace = "com.test.mapper.StudentMapper">
```

```
<!-- sql查询语句映射 -->
<select id = "selectStudent" resultType = "hashmap">
SELECT * FROM student
</select>
</mapper>
```

这里的 com.test.mapper 是 StudentMapper.xml 所在的包名称。

(5) 从 XML 中创建 sqlSessionFactory 实例。

在 classpath 下新建 mybatis-config.xml 文件，在文件中配置 SqlSessionFactory 实例信息，其配置内容如下：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE configuration
PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
<environments default = "development">
<environment id = "development">
<transactionManager type = "JDBC"/>
<!-- 数据库基本信息 -->
<dataSource type = "POOLED">
<property name = "driver" value = "com.mysql.jdbc.Driver"/>
<property name = "url" value = "jdbc:mysql:///mydb"/>
<property name = "username" value = "root"/>
<property name = "password" value = "root"/>
</dataSource>
</environment>
</environments>
<mappers>
<mapper resource = "com/test/mapper/StudentMapper.xml"/>
</mappers>
</configuration>
```

(6) 获取 sqlSession 并执行程序。

新建 Test.java 类代码如下：

```
import java.io.IOException;
import java.io.InputStream;
import java.util.List;
import java.util.Map;
import org.apache.ibatis.io.Resources;
```

```
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
public class Test {
    public static void main(String[] args) {
        //类路径下配置文件名称
        String resource = "mybatis-config.xml";
        InputStream inputStream;
        SqlSession   sqlSession = null ;
        try {
            //配置文件加载
            inputStream = Resources.getResourceAsStream(resource);
            //根据配置文件生成SqlSessionFactory对象
            SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
            //sqlSession获取
            sqlSession = sqlSessionFactory.openSession();
            //执行查询请求
            List<Map> list = sqlSession.selectList("com.test.mapper.StudentMapper.selectStudent");
            //输出查询结果
            for(Map map:list)
            {
                System.out.println(map);
            }
        } catch (IOException e)
        {
            e.printStackTrace();
        }finally
        {
            sqlSession.close();
        }
    }
}
```

运行后其控制台输出结果如下：

```
{stuId = 1, stuName = zhangsan}
{stuId = 2, stuName = lisi}
{stuId = 3, stuName = wangwu}
```

(7) MyBatis 的工作机制。

MyBatis 的工作机制如图 1-2 所示。

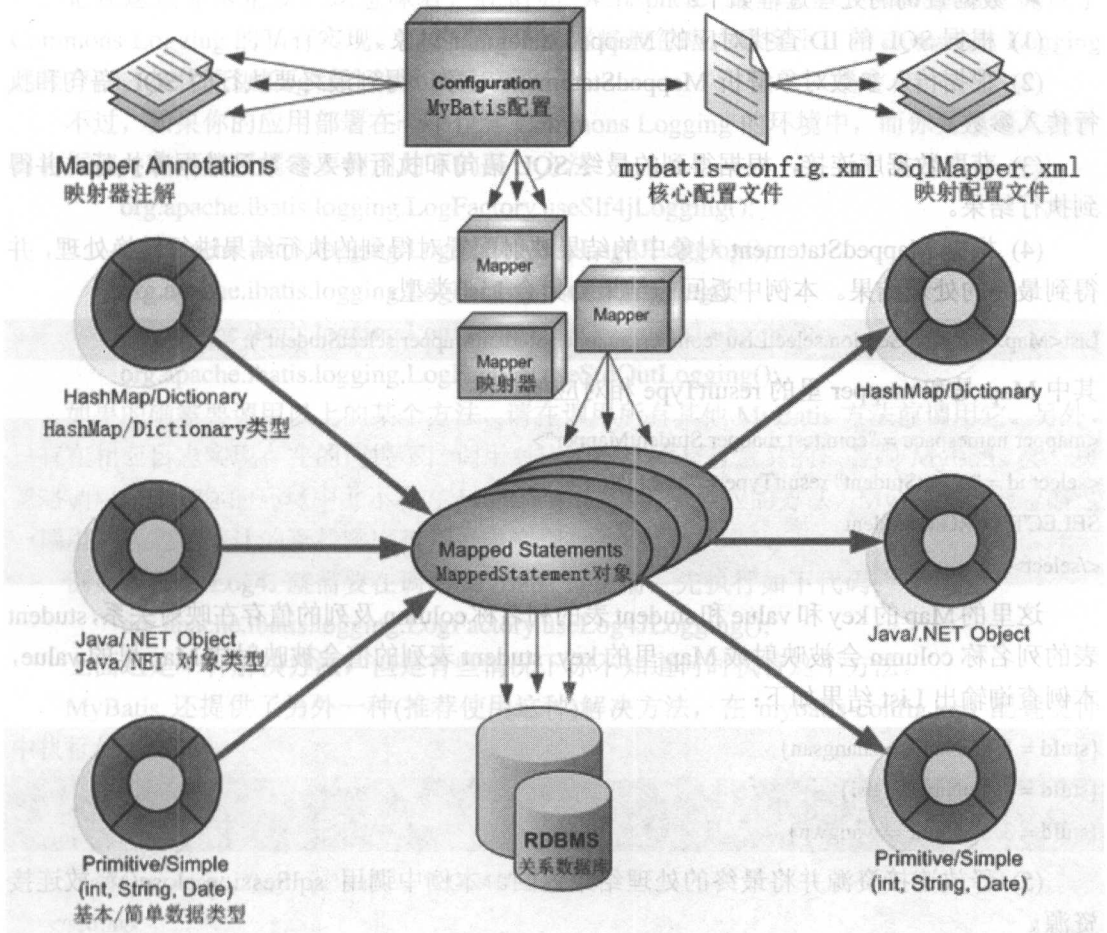

图 1-2　MyBatis 工作机制

➢ 数据查询的具体工作过程如下：

(1) 加载配置文件。

配置来源于两个地方，一处是配置文件(本例中将加载 mybatis-config.xml 和 StudentMapper.xml 配置文件)，一处是 Java 代码的映射器注解(后面章节会讲解)，将 SQL 的配置信息加载成为一个个 MappedStatement 对象(包括了传入参数映射配置、执行的 SQL 语句、结果映射配置)，存储在内存中并接收调用请求。

(2) 调用 MyBatis 提供的 API。

传入参数：SQL 的 ID 和传入参数对象。

处理过程：将请求传递给下层的请求处理层进行处理。

本例中调用 MyBatis 提供的 API 为：

```
List<Map> list = sqlSession.selectList("com.test.mapper.StudentMapper.selectStudent");
```

传入的 SQL 的 ID 为 com.test.mapper.StudentMapper.selectStudent。

(3) 框架操作数据库。

传入参数：SQL 的 ID 和传入参数对象。

➢ 数据查询的处理过程如下：

(1) 根据 SQL 的 ID 查找对应的 MappedStatement 对象。

(2) 根据传入参数对象解析 MappedStatement 对象，得到最终要执行的 SQL 语句和执行传入参数。

(3) 获取数据库连接，根据得到的最终 SQL 语句和执行传入参数到数据库执行，并得到执行结果。

(4) 根据 MappedStatement 对象中的结果映射配置对得到的执行结果进行转换处理，并得到最终的处理结果。本例中返回的结果为集合 List 类型：

```
List<Map> list = sqlSession.selectList("com.test.mapper.StudentMapper.selectStudent");
```

其中 Map 是和 mapper 里的 resultType 相对应的：

```
<mapper namespace = "com.test.mapper.StudentMapper">
<select id = "selectStudent" resultType = "hashmap">
SELECT * FROM student
</select></mapper>
```

这里的 Map 的 key 和 value 和 student 表的列名称 column 及列的值存在映射关系，student 表的列名称 column 会被映射成 Map 里的 key，student 表列的值会被映射成 Map 里的 value，本例查询输出 List 结果如下：

```
{stuId = 1, stuName = zhangsan}
{stuId = 2, stuName = lisi}
{stuId = 3, stuName = wangwu}
```

(5) 释放连接资源并将最终的处理结果返回。本例中调用 sqlSession.close()释放连接资源。

1.4 MyBatis 日志

1.4.1 MyBatis 日志的实现方式

MyBatis 内置的日志工厂提供日志功能，具体的日志实现有以下几种方式：

- SLF4J
- Apache Commons Logging
- Log4J2
- Log4J
- JDK logging

具体选择哪个日志实现由 MyBatis 的内置日志工厂确定。它会使用最先找到的日志(按上文列举的顺序查找)。如果一个都未找到，日志功能就会被禁用。

不少应用服务器的 classpath 中已经包含 Commons Logging，如 Tomcat 和 WebShpere，所以 MyBatis 会把它作为具体的日志实现。

记住这点非常重要。这意味着，在诸如 WebSphere 的环境中，WebSphere 提供了 Commons Logging 的私有实现，你的 Log4J 配置将被忽略。事实上，因 Commons Logging 已经存在，按优先级 Log4J 自然就被忽略了！

不过，如果你的应用部署在一个包含 Commons Logging 的环境中，而你又想用其他的日志框架，你可以根据需要调用如下的某一方法：

```
org.apache.ibatis.logging.LogFactory.useSlf4jLogging();
org.apache.ibatis.logging.LogFactory.useLog4JLogging();
org.apache.ibatis.logging.LogFactory.useJdkLogging();
org.apache.ibatis.logging.LogFactory.useCommonsLogging();
org.apache.ibatis.logging.LogFactory.useStdOutLogging();
```

如果的确需要调用以上的某个方法，请在调用所有其他 MyBatis 方法前调用它。另外，只有在相应日志实现存在的前提下，调用对应的方法才是有意义的，否则 MyBatis 会一概忽略调用。如果你的环境中并不存在 Log4J，你却调用了相应的方法，MyBatis 就会忽略这一调用，代之以默认的查找顺序查找日志实现。

例如，使用 Log4J 就需要在调用 MyBatis 方法前，先执行如下代码：

```
org.apache.ibatis.logging.LogFactory.useLog4JLogging();
```

上面这是一种解决办法，但是有些情况下你不知道何时执行这个方法。

MyBatis 还提供了另外一种(推荐使用这种)解决方法，在 mybatis-config.xml 配置文件中执行如下代码：

```
<configuration>
 <settings>
      <setting name = "logImpl" value = "LOG4J"/>
  </settings>
</configuration>
```

这里只写了关键的一部分配置信息，在你自己配置的基础上增加<setting name = "logImpl" value = "LOG4J"/>即可。这样一来 Log4J 的配置信息就会起作用，value 的取值是 SLF4J、LOG4J、LOG4J2、JDK_LOGGING、COMMONS_LOGGING、STDOUT_ LOGGING、NO_LOGGING 中的一个。

MyBatis 可以对包、类、命名空间和全限定的语句记录日志。

1.4.2 使用 Log4J 实现 MyBatis 日志的配置

【例 1-2】 在例 1-1 的基础上使用 Log4J 方式实现日志输出，具体步骤如下：

步骤 1：增加 Log4J 的 jar 包。

因为采用 Log4J，所以要确保在应用中对应的 jar 包是可用的。要满足这一点，只要将 jar 包添加到应用的 classpath 中即可。Log4J 的 jar 包可以从链接 http://logging.apache.org/log4j/下载。

具体而言，对于 Web 或企业应用，需要将 log4J 的 jar 添加到 WEB-INF/lib 目录下；对于独立应用，可以将它添加到 JVM 的-classpath 启动参数中。

步骤 2：配置 Log4J。

配置 Log4J 比较简单，比如需要记录这个 mapper 命名空间的日志：

```
<mapper namespace = "com.test.mapper.StudentMapper">
<select id = "selectStudent" resultType = "hashmap">
SELECT * FROM student
</select></mapper>
```

只要在应用的 classpath 中创建一个名称为 log4j.properties 的文件，文件的具体内容如下：

```
# Global logging configuration
log4j.rootLogger = ERROR, stdout
# MyBatis logging configuration...
log4j.logger.com.test.mapper.StudentMapper = TRACE
# Console output...
log4j.appender.stdout = org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout = org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern = %5p [%t] - %m%n
```

添加以上配置后，Log4J 就会把 com.test.mapper.StudentMapper 的详细执行日志记录下来，对于应用中的其他类则仅仅记录错误信息。

也可以将日志从整个 mapper 命名空间级别调整到语句级别，从而实现更细粒度的控制。如下配置只记录 selectStudent 语句的日志：

```
log4j.logger.com.test.mapper.StudentMapper.selectStudent = TRACE
```

与此相对，可以对一组 mapper 命名空间记录日志，只要对 mapper 接口所在的命名空间开启日志功能即可：

```
log4j.logger.com.test.mapper.StudentMapper = TRACE
```

某些查询可能会返回大量的数据，只想记录其执行的 SQL 语句该怎么办？为此，MyBatis 中 SQL 语句的日志级别被设为 DEBUG(JDK Logging 中为 FINE)，结果日志的级别为 TRACE(JDK Logging 中为 FINER)。所以，只要将日志级别调整为 DEBUG 即可达到目的：

```
log4j.logger.com.test.mapper.StudentMapper = DEBUG
```

本章小结

在本章中，我们讨论了 MyBatis 的优势，介绍了 MyBaits 访问数据库的优点，还学习了怎样创建一个项目，安装 MyBatis jar 包依赖，创建 MyBatis 配置文件，以及在映射器 Mapper XML 文件中配置 SQL 映射语句，最后，学习了 MyBatis 的日志系统，使用 log4J 实现 MyBatis 日志的配置。

练 习 题

一、选择题

1．在 SSM 框架中，MyBatis 承担的责任是(　　)。

A．定义实体类　　B．数据的增删改查操作

C．业务逻辑的描述　　D．页面展示和控制转发

2．在 MyBatis 中(　　)对象负责数据库的连接、打开、关闭等。

A．sqlSession　　B．sqlSessionFactory

C．sqlConnection　　D．sqlConnectionFactory

3．下列关于 MyBatis 配置文件的说法中错误的是(　　)。

A．所有的标签都必须放在<configuration>标签下

B．配置的标签具有严格的顺序

C．只能配置一个<environment>节点

D．transactionManager 用于指定事务管理器类型

4．关于 MyBatis 的下列说法中错误的是(　　)。

A．MyBatis 本是 apache 的一个开源项目 iBatis

B．2010 年这个项目迁移到了谷歌，从 iBatis 更名为 MyBatis

C．MyBatis 是一个管理请求分发的框架

D．MyBatis 实现了 SQL 语句与代码分离

二、填空题

1．MyBatis 本是_____的一个开源项目，2010 年这个项目迁移到了 Google code，并且改名为 MyBatis。2013 年 11 月迁移到 Github。

2．MyBatis 是支持普通_____查询，_____和_____的优秀_____框架。

3．MyBatis 使用简单的_____或_____进行配置和原始映射，将接口和 Java 的 POJO(Plain Old Java Objects，普通的 Java 对象)映射成数据库中的记录。

4．MyBatis 使将输入的 Java 对象中的_____设置成查询参数和从 _____上生成 Java 对象这两个过程自动化。

5．MyBatis 有内建的_____ 级别的缓存机制，用于缓存 select 语句查询出来的结果。除此之外，MyBatis 还提供了与多种第三方缓存类库的集成支持，如 EHCache、OSCache、Hazelcast。

三、问答题

1．MyBatis 的功能架构分为几层？每一层的作用是什么？

2．简述 MyBatis 的优势。

3．简述 MyBatis 的工作机制。

4．MyBatis 内置的日志工厂提供日志功能，具体的日志实现有哪些？

5．为什么说 Mybatis 是半自动 ORM 映射工具？它与全自动的区别在哪里？

6．简述使用 MyBatis 框架进行数据库编程的步骤。

四、实作题

1．搭建自己的第一个 MyBatise 应用。

2．使用 Log4J 实现 MyBatis 日志的配置。

第二章　配置 MyBatis

在配置 MyBatis 的时候，我们可以通过一个 XML(第一章中用 mybatis-config.xml)来配置，也可以嵌入到其他配置文件中，比如我们后面将要学习的 Spring 配置文件 applicationContext.xml。

本章知识要点

- 使用 XML 方式配置 MyBatis;
- 使用 Java API 方式配置 MyBatis。

2.1　基于 XML 方式配置 MyBatis

MyBatis 的 XML 配置文件包含了影响 MyBatis 行为甚深的设置和属性信息。MyBatis 应用程序主要使用 sqlSessionFactory 实例，一个 sqlSessionFactory 实例可以通过 sqlSessionFactoryBuilder 获得。sqlSessionFactoryBuilder 可以从一个 MyBatis-config.xml 配置文件或者一个预定义的配置类的实例获得。

MyBatis 配置文件中，在标签 configuration 下有多个子标签，其层次结构如下：

configuration
|--- properties(属性)
|--- settings(全局配置参数)
|--- typeAliases(类型别名)
|--- typeHandlers(类型处理器)
|--- environments(环境集合属性对象)
|--- |---environment(环境配置)
|--- |--- |--- transactionManager(事务管理)
|--- |--- |---dataSource(数据源)
|--- mappers(映射器)
|--- objectFactory(对象工厂)
|--- plugins(插件)

如下展示了一个典型的 mybatis-config.xml 中的内容：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE configuration
PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
```

```
<configuration>
    <!-- 属性 -->
    <properties resource = "jdbc.properties">
        <property name = "jdbc.username" value = "root" />
        <property name = "jdbc.password" value = "root" />
    </properties>
    <!-- 全局参数设置 -->
    <settings>
        <setting name = "cacheEnabled" value = "true" />
    </settings>
    <!-- 类型别名 -->
    <typeAliases>
        <typeAlias alias = "student" type = "com.test.domain.Student" />
        <package name = "com.test.domain" />
    </typeAliases>
    <typeHandlers>
        <!-- 使用package元素将扫描 com.test.type 包下的全部类型处理器 -->
        <!-- <package name = "com.test.type"/> -->
        <typeHandler handler = "com.test.type.LocalDateTypeHandler" />
        <typeHandler handler = "com.test.type.LocalTimeTypeHandler" />
    </typeHandlers>
    <environments default = "development">
    <!-- 环境1 -->
        <environment id = "development">
            <transactionManager type = "JDBC" />
    <!-- 数据库信息 -->
            <dataSource type = "POOLED">
                <property name = "driver" value = "${jdbc.driver}" />
                <property name = "url" value = "${jdbc.url}" />
                <property name = "username" value = "${jdbc.username}" />
                <property name = "password" value = "${jdbc.password}" />
            </dataSource>
        </environment>
    <!-- 环境2 -->
        <environment id = "release">
            <transactionManager type = "JDBC" />
    <!-- 数据库信息 -->
            <dataSource type = "POOLED">
                <property name = "driver" value = "com.mysql.jdbc.Driver" />
```

```
                <property name = "url" value = "jdbc:mysql://192.168.1.110:3306/mydb" />
                <property name = "username" value = "root" />
                <property name = "password" value = "root" />
            </dataSource>
        </environment>
    </environments>
    <!-- 映射器 -->
    <mappers>
        <mapper resource = "com/test/mapper/StudentMapper.xml" />
    </mappers>
</configuration>
```

2.1.1　属性 properties

属性配置元素可以将配置值具体化到一个属性文件中，并且使用属性文件的 name 名作为占位符。在上述的配置中，我们将数据库连接属性具体化到文件中，配置 properties 的 resource 指定资源文件名称 jdbc.properties，并且为 driver、url、username、password 属性使用了占位符。

在 jdbc.properties 文件中配置数据库连接参数，如下所示：

```
jdbc.driver = com.mysql.jdbc.Driver
jdbc.url = jdbc:mysql:///mydb
jdbc.username = root
jdbc.password = root
```

在 mybatis-config.xml 文件中为属性使用 jdbc.properties 文件中定义的占位符：

```
<properties resource = "jdbc.properties">
<property name = "jdbc.username" value = "root" />
<property name = "jdbc.password" value = "root" />
</properties>
<dataSource type = "POOLED">
<property name = "driver" value = "${jdbc.driver}"/>
<property name = "url" value = "${jdbc.url}"/>
<property name = "username" value = "${jdbc.username}"/>
<property name = "password" value = "${jdbc.password}"/>
</dataSource>
```

并且，可以在<properties>元素中配置默认参数的值。如果<properties>中定义的元素和属性文件定义的元素的 name 值相同，则它们会被属性文件中定义的值覆盖：

```
<properties resource = "jdbc.properties">
<property name = "jdbc.username" value = "root" />
```

```
<property name = "jdbc.password" value = "root" />
</properties>
```

这里，如果 jdbc.properties 文件包含值 jdbc.username 和 jdbc.password，则上述定义的 username 和 password 的值 root 将会被 jdbc.properties 中定义的对应的 jdbc.username 和 jdbc.password 值覆盖。

2.1.2 全局参数设置 settings

MyBaits 框架运行设置一些全局配置参数(**注意**：设置全局参数会影响 MyBatis 框架的运行，应谨慎设置，比如：开启二级缓存，开启延迟加载等内容)。配置方式如下：

```
<settings>
<setting name = "cacheEnabled" value = "true" />
<setting name = "lazyLoadingEnabled" value = "true" />
<setting name = "multipleResultSetsEnabled" value = "true" />
<setting name = "useColumnLabel" value = "true" />
<setting name = "useGeneratedKeys" value = "false" />
<setting name = "autoMappingBehavior" value = "PARTIAL" />
<setting name = "defaultExecutorType" value = "SIMPLE" />
<setting name = "defaultStatementTimeout" value = "25000" />
</settings>
```

常用全局参数设置内容及说明如表 2-1 所示。

表 2-1 MyBatis 的常用全局参数设置内容

设置参数名称	描　述	有效值	默认值
cacheEnabled	使全局的映射器启用或禁用缓存	true | false	true
lazyLoadingEnabled	全局启用或禁用延迟加载。当禁用时，所有关联对象都会即时加载	true | false	false
aggressiveLazyLoading	当启用时，有延迟加载属性的对象在被调用时将会完全加载任意属性。否则，每种属性将会按需要加载	true | false	true
multipleResultSetsEnabled	允许或不允许多种结果集从一个单独的语句中返回(需要适合的驱动)	true | false	true
useColumnLabel	使用列标签代替列名。不同的驱动在这方面表现不同。参考驱动文档或充分测试两种方法来决定所使用的驱动	true | false	true

续表

设置参数名称	描　　述	有效值	默认值
useGeneratedKeys	允许 JDBC 支持生成的键，需要适合的驱动。如果设置为 true，则这个设置强制生成的键被使用，尽管一些驱动拒绝兼容但仍然有效(比如 Derby)	true | false	false
autoMappingBehavior	指定 MyBatis 如何自动映射列到字段属性。PARTIAL 只会自动映射简单、没有嵌套的结果。FULL 会自动映射任意复杂的结果(嵌套的或其他情况)	NONE | PARTIAL | FULL	PARTIAL
defaultExecutorType	配置默认的执行器。SIMPLE 执行器没有什么特别之处。REUSE 执行器重用预处理语句。BATCH 执行器重用语句和批量更新	SIMPLE| REUSE| BATCH	SIMPLE
defaultStatementTimeout	设置超时时间，它决定驱动等待一个数据库响应的时间	Any positive integer	NotSet (null)
safeRowBoundsEnabled	允许在嵌套语句中使用分页(RowBounds)	true | false	false
mapUnderscoreToCamelCase	是否开启自动驼峰命名规则(camel case)映射，即从经典数据库列名 A_COLUMN 到经典 Java 属性名 aColumn 的类似映射	true | false	false
localCacheScope	MyBatis 利用本地缓存机制(Local Cache)防止循环引用(circular references)和加速重复嵌套查询。默认值为 SESSION，这种情况下会缓存一个会话中执行的所有查询。 若设置值为 STATEMENT，本地会话仅用在语句执行上，对相同 sqlSession 的不同调用将不会共享数据	SESSION | STATEMENT	SESSION
jdbcTypeForNull	当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。某些驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER	JdbcType enumeration.Most common are: NULL, VARCHAR and OTHER	OTHER
lazyLoadTriggerMethods	指定哪个对象的方法触发一次延迟加载	以逗号分隔的方法名	equals, clone, hashCode, toString

2.1.3 类型别名 typeAliases

类型别名是 Java 类型的简称，即 Java 对象的简称，它只是关联到 sqlMapper 配置，配置完该别名后 sqlMapper 中就不必写完整的 Java 对象名称了。在 sqlMapper 配置文件中，对于 resultType 和 parameterType 属性值，我们需要使用 JavaBean 的完全限定名。对表 student 进行条件查询和更新的 sqlMapper 配置代码如下所示：

```
<!-- 查询语句映射 -->
<select id = "selectStudentByStuId" parameterType = "int"
resultType = "com.test.domain.Student">
SELECT * FROM student WHERE stuid = #{stuid}
</select>
<!-- 更新语句映射 -->
<update id = "updateStudent" parameterType = "com.test.domain.Student">
UPDATE student SET stuname = #{stuname} WHERE stuid = #{stuid}
</update>
```

这里我们将 resultType 和 parameterType 属性值设置为 Student 类型的完全限定名 com.test.domain.Student，我们可以为完全限定名取一个别名(alias)，然后在需要使用完全限定名的地方使用别名，而不是到处使用完全限定名。如下所示代码，为完全限定名起一个别名：

```
<typeAliases>
<typeAlias alias = "student" type = "com.test.domain.Student" />
</typeAliases>
```

然后在 sqlMapper 映射文件中，如下所示代码使用 Student 的别名：

```
<!-- 返回类型别名student -->
<select id = "selectStudentByStuId" parameterType = "int"
resultType = "student">
SELECT * FROM student WHERE stuid = #{stuid}
</select>
<!-- 参数类型别名student -->
<update id = "updateStudent" parameterType = "student">
UPDATE student SET stuname = #{stuname} WHERE stuid = #{stuid}
</update>
```

如果 Student.java Bean 定义在 com.test.domain 包中，则 com.test.domain.student 的别名会被注册为 student，代码如下所示：

```
<typeAliases>
<package name = "com.test.domain" />
</typeAliases>
```

还有另外一种方式可为 JavaBeans 起别名，使用注解@Alias，代码如下所示：

```
@Alias("student")
public class Student
{
}
```

注意：@Alias 注解将会覆盖配置文件中的<typeAliases>定义。

除上面的自定义类型别名外，MyBatis 对 Java 的基本类型及其他特殊类型的别名也作了映射关系，如表 2-2 所示。

表 2-2　MyBatis 对 Java 类型的映射

映射别名	Java 的类型
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
byte	Byte
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
object	Object
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

2.1.4 类型处理器 typeHandlers

我们知道 Java 有 Java 的数据类型，数据库有数据库的数据类型，那么我们在往数据库中插入数据的时候 MyBatis 是如何把 Java 类型当做数据库类型插入数据库中，在从数据库读取数据的时候又是如何把数据库类型当做 Java 类型来处理呢？这中间必然要经过一个类型转换。在 MyBatis 中我们可以定义一个叫做 typeHandler 类型处理器的东西，通过它可以实现 Java 类型跟数据库类型的相互转换。MyBatis 定义了一些默认处理器，可以用来设置参数或取结果集时实现自动转换，默认的类型处理器如表 2-3 所示。

表 2-3 默认的类型处理器

类型处理器	Java 类型	JDBC 类型
BooleanTypeHandler	Boolean，boolean	任何兼容的布尔值
ByteTypeHandler	Byte，byte	任何兼容的数字或字节类型
ShortTypeHandler	Short，short	任何兼容的数字或短整型
IntegerTypeHandler	Integer，int	任何兼容的数字和整型
LongTypeHandler	Long，long	任何兼容的数字或长整型
FloatTypeHandler	Float，float	任何兼容的数字或单精度浮点型
DoubleTypeHandler	Double，double	任何兼容的数字或双精度浮点型
BigDecimalTypeHandler	BigDecimal	任何兼容的数字或十进制小数类型
StringTypeHandler	String	CHAR 和 VARCHAR 类型
ClobTypeHandler	String	CLOB 和 LONGVARCHAR 类型
NStringTypeHandler	String	NVARCHAR 和 NCHAR 类型
NClobTypeHandler	String	NCLOB 类型
ByteArrayTypeHandler	byte[]	任何兼容的字节流类型
BlobTypeHandler	byte[]	BLOB 和 LONGVARBINARY 类型
DateTypeHandler	Date(java.util)	TIMESTAMP 类型
DateOnlyTypeHandler	Date(java.util)	DATE 类型
TimeOnlyTypeHandler	Date(java.util)	TIME 类型
SqlTimestampTypeHandler	Timestamp(java.sql)	TIMESTAMP 类型
SqlDateTypeHandler	Date(java.sql)	DATE 类型
SqlTimeTypeHandler	Time(java.sql)	TIME 类型
ObjectTypeHandler	Any	其他或未指定类型
EnumTypeHandler	Enumeration 类型	VARCHAR-任何兼容的字符串类型，作为代码存储(而不是索引)

有些类型 MyBatis 是不支持的，只能自定义类型处理器来处理相应的类型。

【例 2-1】 自定义处理器。

假设表 student 有 createDate、createTime 字段，createDate 类型为 date，createTime 类型为 time，而 JavaBean Student 有一个 LocalDate createDate 的定义，现在我们将对 LocalDate 和 LocalTime 类型自定义处理器，步骤如下：

(1) student 建表语句。

代码如下：

```
CREATE TABLE `student` (
  'stuId' int(10) NOT NULL AUTO_INCREMENT,
  'stuName' varchar(50) DEFAULT NULL,
  'createDate' date,
  'createTime' time,
     PRIMARY KEY (`stuId`)
) ENGINE = InnoDB AUTO_INCREMENT = 1 DEFAULT CHARSET = utf8;
```

(2) Student.java 实体类定义。

代码如下：

```
import java.time.LocalDate;
import java.time.LocalTime;
public class Student {
  private String stuname;
  private int stuid;
  private LocalDate createDate;
  private LocalTime createTime;
  public Student() {
  }
  public LocalTime getCreateTime() {
     return createTime;
  }
  public void setCreateTime(LocalTime createTime) {
     this.createTime = createTime;
  }
  public String getStuname() {
     return stuname;
  }
  public LocalDate getCreateDate() {
     return createDate;
  }
  public void setCreateDate(LocalDate createDate) {
     this.createDate = createDate;
  }
```

```
    public void setStuname(String stuname) {
        this.stuname = stuname;
    }
    public int getStuid() {
        return stuid;
    }
    public void setStuid(int stuid) {
        this.stuid = stuid;
    }
}
```

(3) 新建类型处理器。

有两种做法：实现 org.apache.ibatis.type.TypeHandler 接口，或继承 org.apache.ibatis.type.BaseTypeHandler 类，下面将新建两个类型处理器，分别用这两种方式来实现。

① 支持 java.time.LocalTime 类型的类型处理器 LocalTimeTypeHandler.java，采用的是实现接口的方式。代码如下：

```
package com.test.type;
import java.sql.CallableStatement;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Time;
import java.time.LocalTime;
import org.apache.ibatis.type.JdbcType;
import org.apache.ibatis.type.MappedTypes;
import org.apache.ibatis.type.TypeHandler;
@MappedTypes(LocalTime.class)
//实现TypeHandler接口
public class LocalTimeTypeHandler implements TypeHandler<LocalTime>{
    @Override
    public void setParameter(PreparedStatement ps, int i, LocalTime parameter, JdbcType jdbcType) throws
SQLException {
        if(parameter == null)
        {
            ps.setTime(i, null);
        }
        else
        {
            ps.setTime(i, Time.valueOf(parameter));
```

```
        }
    }
        @Override
        public LocalTime getResult(ResultSet rs, String columnName) throws SQLException {
            Time time = rs.getTime(columnName);
            return time == null ? null : time.toLocalTime();
    }
    @Override
    public LocalTime getResult(ResultSet rs, int columnIndex) throws SQLException {
        Time time = rs.getTime(columnIndex);
        return time == null ? null : time.toLocalTime();
    }
    @Override
    public LocalTime getResult(CallableStatement cs, int columnIndex) throws SQLException {
        Time time = cs.getTime(columnIndex);
        return time == null ? null : cs.getTime(columnIndex).toLocalTime();
    }
}
```

② 支持 java.time.LocalDate 类型的类型处理器 LocalDateTypeHandler.java，采用的是继承类的方式。代码如下：

```
package com.test.type;
import java.sql.CallableStatement;
import java.sql.Date;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.time.LocalDate;
import org.apache.ibatis.type.BaseTypeHandler;
import org.apache.ibatis.type.JdbcType;
import org.apache.ibatis.type.MappedTypes;
@MappedTypes(LocalDate.class)
public class LocalDateTypeHandler extends BaseTypeHandler<LocalDate> {
    @Override
    public void setNonNullParameter(PreparedStatement ps, int i, LocalDate parameter, JdbcType jdbcType)
throws SQLException {
        ps.setDate(i, Date.valueOf(parameter));
    }
    @Override
```

```
public LocalDate getNullableResult(ResultSet rs, String columnName) throws SQLException {
    Date date = rs.getDate(columnName);
    if (date == null)
    {
        return null;
    }
    else
    {
        return date.toLocalDate();
    }
}
@Override
public LocalDate getNullableResult(ResultSet rs, int columnIndex) throws SQLException
{
    Date date = rs.getDate(columnIndex);
    if (date == null)
    {
        return null;
    }
    else
    {
        return date.toLocalDate();
    }
}
@Override
public LocalDate getNullableResult(CallableStatement cs, int columnIndex) throws SQLException {
    Date date = cs.getDate(columnIndex);
    if (date == null)
    {
        return null;
    }
    else
    {
        return date.toLocalDate();
    }
}
}
```

注意：不管是实现的方式还是继承的方式，在设置参数或获取结果集时，都应该考虑

空值的情况，否则在空值上调用方法将抛出异常。

(4) 在 mybatis-config.xml 配置文件中注册新建的类型处理器。

代码如下：

```
<typeHandlers>
   <typeHandler handler = "com.test.type.LocalDateTypeHandler"/>
   <typeHandler handler = "com.test.type.LocalTimeTypeHandler"/>
</typeHandlers>
```

或使用批量注册方式。代码如下：

```
<typeHandlers>
<!-- 使用package元素将扫描 com.test.type 包下的全部类型处理器 -->
        <package name = "com.test.type"/>
 </typeHandlers>
```

注册完成后就可以直接存取数据库里的值并自动转换成相应的类型了。Java 程序向数据库写入 LocalDate 和 LocalTime 类型时，处理器自动将其类型转换成数据库对应的 Date 类型和 Time 类型，Java 程序读取数据库 Date 类型和 Time 类型数据时，处理器自动将其类型转换成 Java 所对应的 LocalDate 和 LocalTime 类型。

2.1.5　环境集合属性对象 environments

假如我们的系统的开发环境和正式环境所用的数据库不一样(这是肯定的)，那么可以设置两个 environment，两个 id 分别对应开发环境(development)和正式环境(release)，则通过配置 environments 的 default 属性就能选择对应的 environment。例如，将 environments 的 deault 属性的值配置为 development，那么系统就会选择 development 的 environment。代码如下：

```
<environments default = "development">
<environment id = "development">
<transactionManager type = "JDBC"/>
<dataSource type = "POOLED">
<property name = "driver" value = "${jdbc.driver}"/>
<property name = "url" value = "${jdbc.url}"/>
<property name = "username" value = "${jdbc.username}"/>
<property name = "password" value = "${jdbc.password}"/>
</dataSource>
</environment>
<environment id = "release">
<transactionManager type = "JDBC"/>
<dataSource type = "POOLED">
```

```
<property name = "driver" value = "com.mysql.jdbc.Driver"/>
<property name = "url" value = "jdbc:mysql://192.168.1.110:3306/mydb"/>
<property name = "username" value = "root"/>
<property name = "password" value = "root"/>
</dataSource>
</environment>
</environments>
```

从上面的配置中可以看到 environments 下还有 environment、dataSource、transactionManager 等三个配置标签，下面将对这三个配置标签分别加以介绍。

1. 环境配置 environment

每个环境集合属性对象 environments 下可以配置多个环境 environment 标签，每个 sqlSessionFactory 实例只能选择其一；如果想连接两个数据库，就需要创建两个 sqlSessionFactory 实例，每个数据库对应一个；而如果是三个数据库，就需要三个实例，以此类推。environment 通过 ID 属性与其他数据库环境相区别。

2. 数据源 dataSource

dataSource 元素使用标准的 JDBC 数据源接口来配置 JDBC 连接对象的资源。

三种内建的数据源类型：

```
type = [ UNPOOLED | POOLED | JNDI ]
```

UNPOOLED：这个数据源的实现只是每次请求时打开和关闭连接。虽然有一点慢，但对在及时和可用连接方面没有性能要求的简单应用程序是一个很好的选择，不同的数据库在这方面的表现也是不一样的，所以对某些数据库来说使用连接池并不重要，这个配置也是理想的。

UNPOOLED 类型的数据源仅仅需要配置以下 5 种属性：

driver：JDBC 驱动的 Java 类的完全限定名；

url：数据库的 JDBC URL 地址；

userName：登录数据库的用户名；

password：登录数据库的密码；

dedaultTransactionIsolationLevel：默认的连接事务隔离级别。

作为可选项，你也可以传递属性给数据库驱动，要这样做，属性的前缀为“driver.”，例如：driver.encoding = UTF8 将通过 DriverManager.getConnection(url, driverProperties)方法传递值为 UTF8 的 encoding 属性给数据库驱动。

POOLED：这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来，避免了创建新的连接实例时所必需的初始化和认证时间。这是一种使得并发 Web 应用快速响应请求的流行的处理方式。

除了上述提到的 UNPOOLED 下的属性外，还有以下属性可用来配置 POOLED 的数据源：

poolMaximumActiveConnections：在任意时间可以存在的活动(正在使用的)连接数量，

默认值为 10。

poolMaximumIdleConnections：任意时间可能存在的空闲连接数。

poolMaximumCheckoutTime：在被强制返回之前，池中连接被再次检出来使用的时间，默认值为 2 万毫秒，即 20 s。

poolTimeToWait：这是一个底层设置，如果获取连接需花费相当长的时间，它会给连接池打印状态日志并重新尝试获取一个连接(避免在误配置的情况下一直安静地失败)，默认值为 2 万毫秒，即 20 s。

PoolPingQuery：发送到数据库的侦测查询，用来检验连接是否处在正常工作秩序中并准备接受请求。默认是“NO PING QUERY SET”，这一设置会使数据库驱动失败时带有一个恰当的错误消息。

PoolPingConnectionsNotUsedFor：配置 poolPingQuery 的使用频度。这可以被设置成匹配具体的数据库连接超时时间，以避免不必要的侦测，默认值为 0 (即所有连接每一时刻都被侦测——仅当 poolPingEnabled 为 true 时适用)。

JDNI：这个数据源的实现是为了能在如 EJB 或应用服务器这类容器中使用，容器可以集中在外部配置数据源，然后放置一个 JDNI 上下文的引用。这种数据源只需要两个属性。initial_context 这个属性用来在 InitailContext 中寻找上下文(即 initialContext.lookup (initial_context))，这是个可选属性，如果忽略，那么 data_source 属性将会直接从 InitialContex 中寻找。

data_source：引用数据源实例位置的上下文的路径。提供了 initial_context 配置时会在其返回的上下文中进行查找，没有提供时则直接在 InitialContext 中查找。

和其他数据源配置类似，可以通过添加前缀“env.”直接把属性传递给初始上下文。比如：

```
env.encoding = UTF8
```

这就是在初始上下文(InitialContext)实例化时向它的构造方法传递值为 UTF8 的 encoding 属性。

数据源配置代码如下：

```
<dataSource type = "POOLED">
<property name = "driver" value = "com.mysql.jdbc.Driver"/>
<property name = "url" value = "jdbc:mysql://192.168.1.110:3306/mydb"/>
<property name = "username" value = "root"/>
<property name = "password" value = "root"/>
</dataSource>
```

3. 事务管理 transactionManager

在 MyBatis 中有两种事务管理器类型(也就是 type =“[JDBC|MANAGED]”)。

JDBC：直接使用了 JDBC 的提交和回滚设置。它依赖于从数据源得到的连接来管理事务范围。

MANAGED：几乎未做什么。它从来不提交或回滚一个连接，而会让容器来管理事务

的整个生命周期(比如 Spring 或 JEE 应用服务器的上下文)，默认情况下它会关闭连接。然而一些容器并不希望这样，因此如果你需要从连接中停止它，将 closeConnection 属性设置为 false 即可。代码如下：

```
<transactionManager type = "MANAGED">
  <property name = "closeConnection" value = "false"/>
</transactionManager>
```

2.1.6 映射器 mappers

上面定义了 MyBatis 的行为的配置元素，下面我们来定义 SQL 映射语句。但是首先我们需要告诉 MyBatis 到哪里去找到这些语句或资源。Java 在自动查找这方面没有提供一个很好的方法，所以最佳的方式是告诉 MyBatis 到哪里去找映射文件。你可以使用相对于类路径的资源引用，或完全限定资源定位符(包括 file:///的 URL)，或类名和包名等。这些配置会告诉 MyBatis 去哪里寻找映射文件，剩下的细节就应该是每个 SQL 映射文件了。通过<mappers>可以用以下四种方式导入 SQL 映射语句：

(1) 通过类路径的相对位置导入 xml 方式的映射文件。配置代码如下：

```
<mappers>
   <mapper resource = "com/test/mapper/BlogMapper.xml"/>
   <mapper resource = "com/test/mapper/StudentMapper.xml"/>
</mappers>
```

(2) 通过文件系统路径或者 WEB URL 地址导入 xml 方式的映射文件。配置代码如下：

```
<mappers>
   <mapper url = "file:///var/com/test/mapper/BlogMapper.xml"/>
   <mapper url = "file:///var/com/test/mapper/StudentMapper.xml"/>
</mappers>
```

(3) 通过后面我们将要讲解的映射接口类的方式导入映射类。配置代码如下：

```
<mappers>
   <mapper class = "com.test.mapper.AuthorMapper"/>
   <mapper class = "com.test.mapper.StudentMapper"/>
</mappers
```

(4) 批量注册指定包下面所有接口映射类。配置代码如下：

```
<mappers>
   <package name = "com.test.mapper"/>
</mappers>
```

2.1.7 对象工厂 objectFactory

MyBatis 每次创建结果对象的新实例时，它都会使用一个对象工厂(objectFactory)实例

来完成。默认的对象工厂需要做的仅仅是实例化目标类，要么通过默认构造方法，要么在参数映射存在的时候通过参数构造方法来实例化。默认情况下，我们不需要配置，MyBatis 会调用默认实现的 objectFactory。除非我们要自定义 objectFactory 的实现，那么我们才需要去手动配置。自定义 objectFactory 只需要去继承 DefaultObjectFactory(是 objectFactory 接口的实现类)并重写其方法即可。这个类用于负责创建对象实体类。从这个类的外部看，其主要作用就是根据一个类的类型得到该类的一个实体对象，比如，我们给它一个 Tiger 的 type，它将会给我们一个 Tiger 的实体对象，我们给它一个 java.lang.List 类型，它将会给我们一个 List 的实体对象。

下面是工厂的实现类从 DefaultObjectFactory 变成了我们自己实现的 ExampleObject Factory，其简单实现的代码如下：

```
package com.test.factory;
import java.util.Collection;
import java.util.Iterator;
import java.util.Properties;
import org.apache.ibatis.reflection.factory.DefaultObjectFactory;
public class ExampleObjectFactory extends DefaultObjectFactory {
    public Object create(Class type)
    {
        if (type.equals(Tiger.class))
        {
            Tiger obj = (Tiger) super.create(type);
            obj.setAge(1000);
            obj.setName("test baynight chen");
            return obj;
        }
        return super.create(type);
    }
    public void setProperties(Properties properties)
    {
        Iterator iterator = properties.keySet().iterator();
        while (iterator.hasNext())
        {
            String keyValue = String.valueOf(iterator.next());
            System.out.println(properties.getProperty(keyValue));
        }
        super.setProperties(properties);
    }
    public <T> boolean isCollection(Class<T> type)
    {
```

```
        return Collection.class.isAssignableFrom(type);
    }
}
```

通过上面代码我们定义了 objectFactory 后，接下来在 mybatis-config.xml 里进行配置，配置代码如下：

```
<objectFactory type = "com.test.factory.ExampleObjectFactory">
    <property name = "someProperty1" value = "1000" />
    <property name = "someProperty2" value = "2000" />
</objectFactory>
```

objectFactory 里几个方法的作用：

Create(Type，type)通过接受一个 type 类型，得到该对象的一个实例，调用的是对象的无参构造函数，内部实现毫无疑问就是使用 Java 的反射，或者是使用了 CGLIB 抑或是 Java 的 ASSIST。

setProperties(Properties properties)方法对在节点中配置的 property 内容进行了加载，可以通过传入的属性文件内容影响程序逻辑，这个与容器对 web.xml 的解析的原理是差不多的，也就是 XML 解析形成对象，然后以参数的方式传递到方法中。

isCollection()方法里通过 Collection.class.isAssignableFrom(type)方式判断要生成的这个对象是不是集合对象，我们应该记住这种判断方式。

2.1.8 插件 plugins

plugins 是一个可选配置。MyBatis 中的 plugin 其实就是个 interceptor，它可以拦截 executor、parameterHandler、resultSetHandler、statementHandler 的部分方法，处理我们自己的逻辑。executor 就是真正执行 SQL 语句的，parameterHandler 是处理我们传入的参数的，在前面讲 typeHandler 时已提到过，MyBatis 默认帮我们实现了不少的 typeHandler，当我们不显示配置 typeHandler 时，MyBatis 会根据参数类型自动选择合适的 typeHandler 执行，其中选择 typeHandler 的工作主要是 ParameterHandler 在做，结果的返回主要是由 resultSetHandler 进行处理。

要自定义一个 plugin，需要去实现 Interceptor 接口。每一个 Interceptor 拦截器都必须实现下面的三个方法：

(1) Object intercept(Invocation invocation)是实现拦截逻辑的地方，内部要通过 invocation. proceed()显式地推进责任链前进，也就是调用下一个拦截器拦截目标方法。

(2) Object plugin(Object target) 就是用当前这个拦截器生成对目标 target 的代理，实际是通过 Plugin.wrap(target, this) 来完成的，把目标 target 和拦截器 this 传给了包装函数。

(3) setProperties(Properties properties)用于设置额外的参数，参数配置在拦截器的 Properties 节点里。

插件写好后也需要在配置文件里进行配置，配置代码如下所示：

```
<plugins>
```

```
    <plugin interceptor = "com.test.plugin.ExamplePlugin">
      <property name = "someProperty" value = "100"/>
    </plugin>
</plugins>
```

2.2　基于 Java API 方式配置 MyBatis

上一节中，我们已经讨论了各种 MyBatis 配置元素，如 envronments、typeAlias 和 typeHandlers，以及如何使用 XML 配置它们。如果你想使用基于 Java API 的 MyBatis 配置，它可以帮你对这些配置元素有更好的理解。在本节中，我们会引用到上一节中描述的一些类。

MyBatis 的 sqlSessionFactory 接口除了使用基于 XML 的配置创建外也可以通过 Java API 编程式地创建。每个在 XML 中配置的元素，都可以编程式地创建。

使用 Java API 创建 sqlSessionFactory，代码如下所示：

```
public static SqlSessionFactory getSqlSessionFactory()
{
    SqlSessionFactory sqlSessionFactory = null;
    try {
        String driver = "com.mysql.jdbc.Driver";
        String url = "jdbc:mysql://localhost:3306/mybatisdemo";
        String username = "root";
        String password = "root";
        DataSource dataSource = new PooledDataSource(driver, url, username,
                password);
        TransactionFactory transactionFactory = new JdbcTransactionFactory();
        Environment environment = new Environment("development",
                transactionFactory, dataSource);
        Configuration configuration = new Configuration(environment);
        configuration.getTypeAliasRegistry().registerAlias("student",
                Student.class);
        configuration.addMapper(StudentMapper.class);
        sqlSessionFactory = new SqlSessionFactoryBuilder()
                .build(configuration);
    }
    catch (Exception e)
    {
        throw new RuntimeException(e);
```

```
    }
    return sqlSessionFactory;
}
```

2.2.1 环境配置 Environment

我们需要为使用 MyBatis 连接的每一个数据库创建一个 Enviroment 对象。为了使用每一个环境，我们需要为每一个环境 Environment 创建一个 sqlSessionFactory 对象。而创建 Environment 对象，我们需要 java.sql.DataSource 和 TransactionFactory 实例。下面让我们看看如何创建 DataSource 和 TransactionFactory 对象。

1. 数据源(DataSource)

MyBatis 支持三种内建的 DataSource 类型：UNPOOLED、POOLED 和 JNDI。

UNPOOLED 类型的数据源 dataSource 为每一个用户请求创建一个数据库连接。在多用户并发应用中，不建议使用。

POOLED 类型的数据源 dataSource 创建了一个数据库连接池，对用户的每一个请求，会使用连接池中的一个可用的 Connection 对象，这样可以提高应用的性能。MyBatis 提供了用 org.apache.ibatis.datasource.pooled.PooledDataSource 实现 javax.sql.DataSource 来创建连接池。

JNDI 类型的数据源 dataSource 使用了应用服务器的数据库连接池，并且使用 JNDI 查找来获取数据库连接。

下面我们来看怎样通过 MyBatis 的 PooledDataSource 获得 DataSource 对象，代码如下：

```
package com.test.factory;
import javax.sql.DataSource;
import org.apache.ibatis.datasource.pooled.PooledDataSource;
public class DataSourceFactory{
    public static DataSource getDataSource(){
        String driver = "com.mysql.jdbc.Driver";
        String url = "jdbc:mysql://localhost:3306/mybatisdemo";
        String username = "root";
        String password = "root";
        PooledDataSource dataSource = new PooledDataSource(driver, url,
        username, password);
        return dataSource;
    }
}
```

一般在生产环境中，DataSource 会被应用服务器配置，并通过 JNDI 获取 DataSource 对象，代码如下所示：

```
package com.test.factory;
import javax.naming.InitialContext;
```

```
import javax.naming.NamingException;
import javax.sql.DataSource;
public class DataSourceFactory
{
    public static DataSource getDataSource()
    {
        String jndiName = "java:comp/env/jdbc/MyBatisDS";
        try
        {
            InitialContext ctx = new InitialContext();
            DataSource dataSource = (DataSource) ctx.lookup(jndiName);
            return dataSource;
        }
        catch (NamingException e)
        {
            throw new RuntimeException(e);
        }
    }
}
```

2. 事务工厂 TransactionFactory

当你将 TransactionManager 属性设置成 JDBC 时，MyBatis 内部将使用 JdbcTransactionFactory 类创建 TransactionManager，代码如下所示：

```
DataSource dataSource = DataSourceFactory.getDataSource();
TransactionFactory txnFactory = new JdbcTransactionFactory();
```

当你将 TransactionManager 属性设置成 MANAGED 时，MyBatis 内部将使用 ManagedTransactionFactory 类创建事务管理器 TransactionManager，代码如下所示：

```
DataSource dataSource = DataSourceFactory.getDataSource();
TransactionFactory txnFactory = new ManagedTransactionFactory();
```

最后，Enviroment 对象的创建依赖于上面创建的数据源和事务工厂，代码如下所示：

```
Environment environment = new Environment("development", txnFactory,
dataSource);
```

2.2.2 类型别名 typeAliases

MyBatis 提供以下几种通过 Configuration 对象注册类型别名的方法：

(1) 根据默认的别名规则，使用一个类的首字母小写、非完全限定的类名作为别名注册，可使用以下代码：

```
configuration.getTypeAliasRegistry().registerAlias(Student.class);
```

(2) 指定别名注册，可使用如下代码：

```
configuration.getTypeAliasRegistry().registerAlias("Student", Student.class);
```

(3) 通过类的完全限定名注册相应类别名，可使用如下代码：

```
configuration.getTypeAliasRegistry().registerAlias("Student",
"com.test.domain.Student");
```

(4) 为某一个包中的所有类注册别名，可使用如下代码：

```
configuration.getTypeAliasRegistry().registerAliases("com.test.domain");
```

(5) 为在 com.test.domain package 包中所有的继承自 Identifiable 类型的类注册别名，可使用如下代码：

```
configuration.getTypeAliasRegistry().registerAliases("com.
test.domain", Identifiable.class);
```

2.2.3 类型处理器 typeHandlers

MyBatis 提供了一系列使用 Configuration 对象注册类型处理器(typehandler)的方法。我们可以通过以下方式注册自定义的类处理器：

(1) 为某个特定的类注册类处理器，代码如下所示：

```
configuration.getTypeHandlerRegistry().register(MyDate.
class, DateTypeHandler.class);
```

(2) 注册一个类处理器，代码如下所示：

```
configuration.getTypeHandlerRegistry().register(MyDateTypeHandler.
class);
```

(3) 注册 com.test.typehandlers 包中的所有类型处理器，代码如下所示：

```
configuration.getTypeHandlerRegistry().register("com.test.typehandlers");
```

2.2.4 全局参数设置 Settings

MyBatis 提供了一组默认的、能够很好地适用大部分应用的全局参数设置。然而，你可以稍微调整这些参数，让它更好地满足应用的需要。你可以使用下列方法将全局参数设置成想要的值，代码如下所示：

```
configuration.setCacheEnabled(true);
configuration.setLazyLoadingEnabled(false);
configuration.setMultipleResultSetsEnabled(true);
configuration.setUseColumnLabel(true);
configuration.setUseGeneratedKeys(false);
configuration.setAutoMappingBehavior(AutoMappingBehavior.PARTIAL);
```

```
configuration.setDefaultExecutorType(ExecutorType.SIMPLE);
configuration.setDefaultStatementTimeout(25);
configuration.setSafeRowBoundsEnabled(false);
configuration.setMapUnderscoreToCamelCase(false);
configuration.setLocalCacheScope(LocalCacheScope.SESSION);
configuration.setAggressiveLazyLoading(true);
configuration.setJdbcTypeForNull(JdbcType.OTHER);
lazyLoadTriggerMethods.add("equals");
lazyLoadTriggerMethods.add("clone");
lazyLoadTriggerMethods.add("hashCode");
lazyLoadTriggerMethods.add("toString");
Set<String> lazyLoadTriggerMethods = new HashSet<String>();
configuration.setLazyLoadTriggerMethods(lazyLoadTriggerMethods );
```

2.2.5　映射器 mappers

MyBatis 提供了一些使用 Configuration 对象注册 mapper XML 文件和 mappe 接口的方法，如下所述：

(1) 添加一个 mapper 接口，可使用以下代码：

```
configuration.addMapper(StudentMapper.class);
```

(2) 添加 com.test.mapper 包中的所有 mapper XML 文件或者 mapper 接口，可使用以下代码：

```
configuration.addMappers("com.test.mapper");
```

(3) 添加所有 com.test.mapper 包中的拓展了特定 mapper 接口的 mapper 接口，如 BaseMapper，可使用如下代码：

```
configuration.addMappers("com.test.mapper", BaseMapper.class);
```

注意：mappers 应该在 typeAliases 和 typeHandler 注册后再添加到 configuration 中。

本章小结

在本章中我们学习了怎样使用 XML 方式和基于 Java API 方式配置 MyBatis。在介绍 XML 方式引导 MyBatis 时，我们学习了 properties 属性、settings 全局参数、typeAliases 类型别名、typeHandlers 类型处理器、environments 环境集合属性对象、mappers 映射器、objectFactory 对象工厂和 plugins 插件的 XML 配置使用方式，在介绍 Java API 方式引导 MyBatis 时，我们学习了 environment 环境配置、typeAliases 类型别名、typeHandlers 类型处理器、settings 全局参数和 mappers 映射器通过 Java API 的使用方式。

练 习 题

一、选择题

1. 关于 MyBatis 的动态标签 trim 的说法错误的是(　　)。

A. 去掉包含元素的空格　　B. 在包含的元素前后加上指定字符

C. 拥有 prefix 属性　　D. 拥有 suffix 属性

2. 针对下列关于 MyBatis 配置文件的两种说法正确的选择是(　　)。

(1) 所有的标签都必须放在<configuration>标签下

(2) 配置的标签具有严格的顺序

A. 都正确　　B. 只有(1)正确

C. 只有(2)正确　　D. 都不正确

二、填空题

1. 属性 properties 配置元素可以将配置值具体化到一个_____中，并且使用属性文件的 name 名作为占位符。

2. 你可以在<properties>元素中配置默认参数的值。如果<properties>中定义的元素和属性文件定义元素的_____值相同，它们会被属性文件中定义的值覆盖。

3. 全局参数会影响 MyBatis 框架运行，需谨慎设置，比如：_____是配置使全局的映射器启用或禁用缓存。

4. 全局启用或禁用延迟加载是使用_____设置。

5. 使用_____设置，有延迟加载属性的对象在被调用时将会完全加载任意属性；否则，每种属性将会按需要加载。

6. MyBatis 使用_____来定义类型别名。

7. 使用_____ 注解定义类型别名将会覆盖配置文件中的<typeAliases>定义。

8. MyBatis 对 Java 的基本类型及其他特殊类型的别名也作了映射关系，映射别名_____和_____分别对应着 Java 的类型 String 和 int。

9. MyBatis 使用_____进行类型处理器定义。

10. environment 标签对应着每一个环境的配置，每个环境集合属性对象_____下可以配置多个环境 environment 标签。

11. 我们系统的开发环境和正式环境所用的数据库不一样(这是肯定的)，那么可以设置两个 environment，两个_____分别对应开发环境(development)和正式环境(release)。

12. dataSource 元素使用标准的_____数据源接口来配置 JDBC 连接对象的资源。

13. dataSource 元素三种内建的数据源类型分别是_____、_____、_____。

14．在 MyBatis 中有两种事务管理器类型，分别是_____、_____。

15．MyBatis 全局配置文件中，使用_____告诉 MyBatis 到哪里去找映射文件。

16．MyBatis 每次创建结果对象的新实例时，它都会使用一个_____实例来完成。

17．默认的对象工厂需要做的仅仅是实例化目标类，要么通过_____方法，要么在参数映射存在的时候通过_____方法来实例化。

三、问答题

1．简述<mappers>导入映射的四种方式。

2．简述 API 方式配置映射器 mappers 的方式。

3．MyBatis 中的 association 标签和 collection 标签的作用是什么？有什么区别？

第三章　映射器(Mapper)

MyBatis 真正强大之处就在它的映射语句上。如果比较 SQL 映射配置与 JDBC 代码，可以发现，使用 SQL 映射配置可以节省很大的代码量。MyBatis 映射配置主要被用来创建 SQL 语句，但又给自己的实现预留有极大的空间。在代码里直接嵌套 SQL 语句是很差的编码实践，并且维护起来也比较困难。MyBaits 使用了映射器配置文件或注解来配置 SQL 语句，使 SQL 语句和代码分离，极大地提高了代码的后期可维护性。

本章知识要点

- SQL 映射配置文件和 SQL 映射接口；
- SQL 映射；
- SQL 高级映射；
- 动态 SQL；
- 注解配置 SQL 映射器。

3.1　SQL 映射配置文件和 SQL 映射接口

在前两章中我们已经用过一些在映射文件中配置基本的 SQL 映射语句，以及传入条件和返回结果，最后使用 sqlSession 对象调用它们的例子，其相关的代码片段如下：

Student 表结构及样本数据：

```
CREATE TABLE 'student' (
    'stuId' int(10) NOT NULL AUTO_INCREMENT,
    'stuName' varchar(50) DEFAULT NULL,
    PRIMARY KEY ('stuId')
) ENGINE = InnoDB AUTO_INCREMENT = 1 DEFAULT CHARSET = utf8;
INSERT INTO 'student' VALUES ('1', 'zhangsan');
INSERT INTO 'student' VALUES ('2', 'lisi');
INSERT INTO 'student' VALUES ('3', 'wangwu');
```

com.test.mapper 包下 StudentMapper.xml 映射文件：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
```

```
<mapper namespace = "com.test.mapper.StudentMapper">
<select id = "selectStudent" resultType = "hashmap">
SELECT * FROM student
</select>
</mapper>
```

mybatis-config.xml 配置文件内容如下：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE configuration
PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
<environments default = "development">
<environment id = "development">
<transactionManager type = "JDBC"/>
<dataSource type = "POOLED">
<property name = "driver" value = "com.mysql.jdbc.Driver"/>
<property name = "url" value = "jdbc:mysql:///mydb"/>
<property name = "username" value = "root"/>
<property name = "password" value = "root"/>
</dataSource>
</environment>
</environments>
<mappers>
<mapper resource = "com/test/mapper/StudentMapper.xml"/>
</mappers>
</configuration>
```

测试调用类 Test.java：

```
import java.io.IOException;
import java.io.InputStream;
import java.util.List;
import java.util.Map;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
public class Test {
    public static void main(String[] args) {
        String resource  =  "mybatis-config.xml";
```

```
        InputStream inputStream;
        SqlSession   sqlSession = null ;
        try {
             //配置文件加载
             inputStream = Resources.getResourceAsStream(resource);
            //根据配置文件生成SqlSessionFactory对象
            SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
            //sqlSession获取
            sqlSession = sqlSessionFactory.openSession();
            //执行查询请求
            List<Map> list = sqlSession.selectList("com.test.mapper.StudentMapper.selectStudent");
            //输出查询结果
            for(Map map:list)
            {
                System.out.println(map);
            }
        } catch (IOException e)
        {
            e.printStackTrace();
        }finally
        {
            sqlSession.close();
        }
    }
}
```

现在让我们在 com.test.mapper 包中的 StudentMapper.xml 配置文件内，配置 id 为“selectStudentByStuId”和“updateStudent”的 SQL 语句的映射，代码如下：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace = "com.test.mapper.StudentMapper">
<select id = "selectStudentByStuId" parameterType = "int"
resultType = "student">
SELECT * FROM student WHERE stuid = #{stuid}
</select>
<update id = "updateStudent" parameterType = "student">
UPDATE student SET stuname = #{stuname} WHERE stuid = #{stuid}
```

```
</update>
</mapper>
```

映射文件配置好了之后，在代码里我们就可以通过 sqlSession 进行调用了，selectStudentByStuId 调用，代码如下：

```
String resource = "mybatis-config.xml";
    InputStream inputStream;
    SqlSession sqlSession = null;
    try {
        //加载配置文件
        inputStream = Resources.getResourceAsStream(resource);
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder()
                .build(inputStream);
        sqlSession = sqlSessionFactory.openSession();
        Student student = new Student();
        List<Student> list = sqlSession.selectList(
                "com.test.mapper.StudentMapper.selectStudentByStuId", 1);
        for (Student stu : list)
        {
            System.out.println(stu.getStuname());
        }
    } catch (Exception e)
    {
        e.printStackTrace();
    } finally
    {
    sqlSession.close();
}
```

updateStudent 调用，代码如下：

```
String resource = "mybatis-config.xml";
    InputStream inputStream;
    SqlSession sqlSession = null;
    try {
        inputStream = Resources.getResourceAsStream(resource);
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder()
                .build(inputStream);
        sqlSession = sqlSessionFactory.openSession();
        Student student = new Student();
        student.setStuid(1);
```

```
        student.setStuname("cyz");
        int rows = sqlSession.update("com.test.mapper.StudentMapper.updateStudent",
        student);
        System.out.println(rows);
        sqlSession.commit();
    } catch (Exception e)
    {
        sqlSession.rollback();
        e.printStackTrace();
    } finally
    {
        sqlSession.close();
    }
```

从上面的例子中我们可以发现要调用已经配置好的映射语句可以通过字符串(字符串形式为映射器配置文件所在的包名 namespace+在文件内定义的语句 id，如上，即 com.test.mapper.StudentMapper 和语句 id selectStudentByStuId 或 updateStudent 组成)调用映射的 SQL 语句，但是这种方式虽然简洁却容易出错。在调用之前必须要检查映射器配置文件中的定义，以保证输入参数类型和结果返回类型是有效的；否则可能会产生运行时异常。

为了避免可能产生的运行时异常，MyBatis 提供了映射接口调用方式，一旦通过映射器配置文件配置了映射语句，就可以创建一个完全对应的映射器接口，接口名跟配置文件名相同，接口所在包名也跟配置文件所在包名完全一样(如 StudentMapper.xml 所在的包名是 com.test.mapper，对应的接口名就是 com.test.mapper.StudentMapper.java)。映射器接口中的方法签名也跟映射器配置文件中完全对应：方法名为配置文件中的 id 值；方法参数类型为 parameterType 对应值；方法返回值类型为 returnType 对应值。对于上述的 StudentMapper.xml 文件，我们可以创建一个映射器接口 StudentMapper.java，代码如下所示：

```
package com.test.mapper;
import java.util.List;
import com.test.domain.Student;
public interface StudentMapper {
    public List<Student> selectStudentByStuId(Integer stuid);
    public int updateStudent(Student student);
}
```

在 StudentMapper.xml 映射器配置文件中，其名字空间 namespace 应该跟 StudentMapper 接口的完全限定名保持一致。另外，StudentMapper.xml 中语句 id、parameterType、returnType 应该分别和 StudentMapper 接口中的方法名、参数类型、返回值相对应。

使用映射器接口可以以类型安全的形式调用映射语句。调用代码如下所示：

```
String resource = "mybatis-config.xml";
```

```
InputStream inputStream;
SqlSession sqlSession = null;
try {
    inputStream = Resources.getResourceAsStream(resource);
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder()
                        .build(inputStream);
    sqlSession = sqlSessionFactory.openSession();
    Student student = new Student();
    student.setStuid(1);
    student.setStuname("cyz");
    StudentMapper studentMapper = sqlSession.getMapper(StudentMapper.class);
    List<Student> list = studentMapper.selectStudentByStuId(1);
    for (Student stu : list) {
        System.out.println(stu.getStuname());
    }
    int rows = studentMapper.updateStudent(student);
    System.out.println(rows);
    sqlSession.commit();
} catch (Exception e)
{
    sqlSession.rollback();
    e.printStackTrace();
} finally
{
    sqlSession.close();
}
```

3.2　SQL 映射

对于 SQL 映射 XML 文件 MyBatis 提供了一些基本的配置标签，如下所列：

- cache——在特定的命名空间配置缓存。
- cache-ref——引用另外一个命名空间配置的缓存。
- resultMap——最复杂也是最强大的元素，用来描述如何从数据库结果集里加载对象。
- SQL——能够被其他语句重用的 SQL 块。
- insert——insert 映射语句。
- update——update 映射语句。
- delete——deleete 映射语句。
- select——select 映射语句。

3.2.1 select 查询语句

一个 select SQL 语句可以在<select>元素的映射器 XML 配置文件中配置，代码如下所示：

```
<select id = "selectStudentByStuId" parameterType = "int"
        resultType = "student">
        SELECT * FROM student WHERE stuid = #{stuid}
</select>
```

上面这条语句叫做 selectStudentByStuId 语句，它以 int 型(或者 Integer 型)作为输入参数，并返回一个 student(类型别名 typeAliases)类型的值。注意这个参数的表示法：#{stuid}，它告诉 MyBatis 生成 PreparedStatement 参数。像在 JDBC 里，这个参数会被标识为“?”，然后传递给 PreparedStatement，代码如下：

```
String selectStudent = "SELECT * FROM student WHERE stuid = ?" ;
PreparedStatement ps = conn.prepareStatement(selectStudent);
ps.setInt(1, stuid);
```

当然，如果单独使用 JDBC 去提取这个结果集并把结果集映射到对象上，则需要更多的代码，而这些，MyBatis 都已经做到了。

上面映射使用一个 ID selectStudentByStuId，可以在名空间 com.test.mapper.StudentMapper. selectStudentByStuId 中唯一标识。

如果我们要根据学生 ID 得到学生信息列表，可以如下调用代码：

```
List<Student> list = sqlSession.selectList(
                    "com.test.mapper.StudentMapper.selectStudentByStuId", 1);
```

sqlSession.selectList()方法返回执行 select 语句后所返回的 List<Student>的值。

如果不使用名空间(namespace)和语句 id 来调用映射语句，你可以通过创建一个映射器 mapper 接口，并以类型安全的方式调用方法，代码如下所示：

```
package com.test.mapper;
import java.util.List;
import com.test.domain.Student;
public interface StudentMapper {
    public List<Student> selectStudentByStuId(Integer stuid);
}
```

这样，根据学生 ID 得到学生信息列表，可以用如下代码调用 selectStudentByStuId 映射语句实现：

```
StudentMapper studentMapper = sqlSession.getMapper(StudentMapper.class);
List<Student> list = studentMapper.selectStudentByStuId(1);
```

select 语句还有很多的属性允许您详细配置每一条语句，配置代码如下所示：

```
<select
    id = "selectStudentByStuId"
    parameterType = "int"
    parameterMap = "deprecated"
    resultType = "hashmap"
    resultMap = "personResultMap"
    flushCache = "false"
    useCache = "true"
    timeout = "10000"
    fetchSize = "256"
    statementType = "PREPARED"
    resultSetType = "FORWARD_ONLY"
>
```

select 每个属性都代表着不同的含义，select 语句的详细配置属性解释如表 3-1 所示。

表 3-1　select 语句详细配置属性解释

属性名	描　　述
id	在这个命名空间下唯一的标识符，可被其他语句引用
parameterType	传给此语句的参数的完整类名或别名
parameterMap	不推荐使用。这个参数将来可能被删除
resultType	语句返回值类型的完整类名或别名。注意，如果返回的是集合(collections)，那么应该是集合所包含的具体子类型，而不是集合本身。resultType 与 resultMap 不能同时使用
resultMap	引用外部定义的 resultMap。结果集映射是 MyBatis 中最强大的特性，同时又非常好理解。许多复杂的映射都可以轻松解决。resultType 与 resultMap 不能同时使用
flushCache	如果设为 true，则在每次语句调用的时候会清空缓存。select 语句默认设为 false
useCache	如果设为 true，则语句的结果集将被缓存，select 语句默认设为 false
timeout	设置超时时间，默认没有设置，由驱动器自己决定
fetchSize	设置从数据库获得记录的条数，默认没有设置，由驱动器自己决定
statementType	可设置为 Statement，Prepared 或 Callable 中的任意一个，告诉 MyBatis 分别使用 Statement，PreparedStatement 或者 CallableStatement。默认为 Prepared
resultSetType	FORWARD_ONLY、SCROLL_SENSITIVE、SCROLL_INSENSITIVE 三个中的任意一个，默认没有设置，由驱动器自己决定

3.2.2 insert 插入语句

1. insert 语句配置

要向 Student 学生表插入一条信息，我们可以使用 insert 语句实现，一个 insert SQL 语句可以在<insert>元素的映射器 XML 配置文件中配置，代码如下所示：

```
<insert id = "insertStudent" parameterType = "student">
INSERT INTO student(stuid, stuname) VALUES(#{stuid}, #{stuname})
</insert>
```

这里我们使用一个 ID insertStudent，可以在名空间 com.test.mapper.StudentMapper.insertStudent 中唯一标识。parameterType 属性应该是一个完全限定类名或者是一个类型别名(typeAliases)。我们可以调用如下这个语句实现向 student 表插入一条学生信息：

```
int rows = sqlSession.insert("com.test.mapper.StudentMapper.insertStudent", student);
```

上面代码中 sqlSession.insert()方法返回执行 insert 语句后插入的行数。

如果不使用命名空间(namespace)加映射语句 id 来调用映射语句，你可以通过创建一个映射器 mapper 接口，并以类型安全的方式来调用方法，代码如下所示：

```
package com.test.mapper;
import java.util.List;
import com.test.domain.Student;
public interface StudentMapper {
    public int insertStudent(Student student);
}
```

你可以调用如下 insertStudent 映射语句：

```
StudentMapper studentMapper = sqlSession.getMapper(StudentMapper.class);
int rows = studentMapper.insertStudent(student);
```

对于 insert 语句的配置，MyBatis 还提供很多的属性允许详细配置每一条语句，配置代码如下所示：

```
<insert
    id = "insertStudent"
    parameterType = "student"
    flushCache = "true"
    statementType = "PREPARED"
    keyProperty = ""
    useGeneratedKeys = ""
    timeout = "20000"
>
```

insert 每个属性都代表着不同的含义，insert 语句详细配置解释如表 3-2 所示。

表 3-2　insert 语句的详细配置解释

属性名	描　　述
id	在这个命名空间下唯一的标识符，可被其他语句引用
parameterType	传给此语句的参数的完整类名或别名
parameterMap	不推荐使用，将来可能删除
flushCache	如果设为 true，则在每次语句调用的时候会清空缓存。select 语句默认设为 false
timeout	设置超时时间，默认没有设置，由驱动器自己决定
statementType	可设置为 STATEMENT、PREPARED 或 CALLABLE 中的任意一个，告诉 MyBatis 分别使用 Statement、PreparedStatement 或者 CallableStatement。默认为 PREPARED
useGeneratedKeys	(仅限 insert 语句时使用)告诉 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来获取数据库自动生成主键(如 MySQL、SQLSERVER 等关系型数据库会有自增的字段)，默认为 false
keyProperty	(仅限 insert 语句时使用)设置自动生成主键的字段，这个字段的值由 getGeneratedKeys 方法返回，或者由 insert 元素的 selectKey 子元素返回。默认不设置

2. 自动生成主键

在上述的学生信息插入 insert 语句中，我们可以为列名 stuid 自动插入值。我们可以使用 useGeneratedKeys 和 keyProperty 属性让数据库生成 auto_increment 列的值，并将生成的值设置到其中一个输入对象属性内，代码如下所示：

```
<insert id = "insertStudent" parameterType = "student" useGeneratedKeys = "true"
    keyProperty = "stuid">
    INSERT INTO student(stuname) VALUES(#{stuname})
</insert>
```

这里 stuid 列值将会被 MySQL 数据库自动生成，并且生成的值会被设置到 student 对象的 stuid 属性上，插入代码如下：

```
StudentMapper studentMapper = sqlSession.getMapper(StudentMapper.class);
studentMapper.insertStudent(student);
```

现在可以用如下代码获取插入的 student 记录的 stuid 的值：

```
int studentId = student.getStuid();
```

有些数据库如 Oracle 并不支持 AUTO_INCREMENT 列，而使用序列(SEQUENCE)来生成主键值。假设我们有一个名为 STU_ID_SEQ 的序列来生成 stuid 主键值。使用如下代码来生成主键：

```
<insert id = "insertStudent" parameterType = "student">
<selectKey keyProperty = "stuid" resultType = "int" order = "BEFORE">
SELECT ELEARNING.STU_ID_SEQ.NEXTVAL FROM DUAL
</selectKey>
```

```
    INSERT INTO student(stuid, stuname)
    VALUES(#{stuid}, #{stuname})
</insert>
```

<selectKey>语句还有其他的属性，每个属性所表达的意义，如表 3-3 所示。

表 3-3 <selectKey>语句属性所表达的意义

属 性	描 述
keyProperty	设置需要自动生成键值的列
resultType	结果类型，MyBatis 通常可以自己检测到，但这并不影响给它一个确切的类型。MyBatis 允许使用任何基本的数据类型作为键值，也包括 String 类型
order	可以设成 BEFORE 或者 AFTER，如果设为 BEFORE，那它会先选择主键，然后设置 keyProperty，再执行 insert 语句；如果设为 AFTER，它就先执行 insert 语句再执行 selectKey 语句，像数据库 Oracle 那样在 insert 语句中调用内嵌的序列机制一样
statementType	像前面一样，MyBatis 支持 STATEMENT、PREPARED 和 CALLABLE 语句类型，分别对应 Statement、PreparedStatement 和 CallableStatement

这里我们使用了<selectKey>子元素来生成主键值，并将值保存到 student 对象的 stuid 属性上。属性 order = “before” 表示 MyBatis 将取得序列的下一个值作为主键值，并且在执行 insert SQL 语句之前将值设置到 stuid 属性上。

我们也可以在获取序列的下一个值时，使用触发器(trigger)来设置主键值，并且在执行 insert SQL 语句之前将值设置到主键列上。如果你采取这样的方式，则对应的 insert 映射语句代码如下所示：

```
<insert id = "insertStudent" parameterType = "student">
    INSERT INTO student(stuname)
    VALUES(#{stuname})
    <selectKey keyProperty = "stuid" resultType = "int" order = "AFTER">
        SELECT ELEARNING.STU_ID_SEQ.CURRVAL FROM DUAL
    </selectKey>
</insert>
```

3.2.3 update 修改语句

一个实现修改学生信息的 update SQL 语句可以在<update>元素的映射器 XML 配置文件中配置，配置代码如下所示：

```
<update id = "updateStudent" parameterType = "student">
    UPDATE student SET stuname = #{stuname} WHERE stuid = #{stuid}
</update>
```

对于 update 语句的配置 MyBatis 还提供有很多的属性允许详细配置每一条语句：

```
<update
    id = "updateStudent"
```

```
    parameterType = "student"
    flushCache = "true"
    statementType = "PREPARED"
    timeout = "20000"
>
```

update 语句的详细配置属性意义和 insert 语句的相同属性表达的意义相同，这里就不再阐述。

在调用映射语句时可以用如下代码：

```
int rows = sqlSession.update("com.test.mapper.StudentMapper.insertStudent", student);
```

sqlSession.update()方法返回执行 update 语句之后影响的行数。

如果不使用命名空间(namespace)和语句 id 来调用映射语句，你可以通过创建一个映射器 mapper 接口，并以类型安全的方式调用方法，代码如下所示：

```
package com.test.mapper;
import java.util.List;
import com.test.domain.Student;
public interface StudentMapper {
    public int updateStudent(Student student);
}
```

可以使用映射器 Mapper 接口来调用 updateStudent 语句，代码如下所示：

```
StudentMapper studentMapper = sqlSession.getMapper(StudentMapper.class);
int rows = studentMapper.updateStudent(student);
```

3.2.4　delete 删除语句

一个 delete SQL 语句可以在<delete>元素的映射器 XML 配置文件中配置，配置代码如下所示：

```
<delete id = "deleteStudent" parameterType = "int">
    DELETE FROM student WHERE stuid = #{stuid}
</delete>
```

对于 delete 语句的配置，MyBatis 还提供有很多的属性允许详细配置每一条语句：

```
<delete
    id = "deleteStudent"
    parameterType = "student"
    flushCache = "true"
    statementType = "PREPARED"
    timeout = "20000"
>
```

delete 语句的详细配置属性意义和 insert 语句相同属性表达的意义相同，这里就不再阐述。

可以用如下代码调用此映射语句：

```
int stuid = 1;
int rows =
sqlSession.delete("com.test.mapper.StudentMapper.deleteStudent", stuid);
```

sqlSession.delete()方法返回 delete 语句执行后影响的行数。

如果不使用命名空间(namespace)和语句 id 来调用映射语句，你可以通过创建一个映射器 mapper 接口，并以类型安全的方式调用方法，代码如下所示：

```
package com.test.mapper;
import java.util.List;
import com.test.domain.Student;
public interface StudentMapper {
    public int deleteStudent(Integer stuid);
}
```

你可以使用映射器 mapper 接口来调用 deleteStudent 语句，代码如下所示：

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
int rows = mapper.deleteStudent(stuid);
```

3.2.5 SQL 块语句

SQL 元素用来定义能够被其他语句引用的可重用 SQL 语句块。例如：

```
<sql id = "userColumns" > id, username, password </sql>
```

这个 SQL 语句块能够被其他语句引用，配置代码如下：

```
<select id = "selectUser" parameterType = "int" resultType = "hashmap" >
select <include refid = "userColumns" />
    from userinfo where id = #{id}
</select>
```

3.2.6 Parameters 参数

前面的语句中，我们看到一些例子中简单的参数(用于代入 SQL 语句中的可替换变量，如#{stuid})。参数是 MyBatis 中非常强大的配置属性，基本上，大部分的情况都会用到，代码如下：

```
<select id = "selectStudentByStuId" parameterType = "int"
        resultType = "student">
        SELECT * FROM student WHERE stuid = #{stuid}
    </select>
```

这段代码演示了一个非常简单的命名参数映射，parameterType 被设置为“int”。参数可以设置为任何类型。像基本数据类型或者像 Integer 和 String 这样的简单的数据对象，(因为没有相关属性)将使用全部参数值。而如果传递的是复杂对象(一般是指 JavaBean)，那么情况就有所不同。代码如下：

```
<insert id = "insertStudent" parameterType = "student">
    INSERT INTO student(stuid, stuname) VALUES(#{stuid}, #{stuname})
</insert>
```

如果参数对象 student 被传递给 SQL 语句，那它将会搜寻 PreparedStatement 里的 stuid 和 stuname 属性，并被 student 对象里相应的属性值替换。

这种传递参数到语句的方式非常简单。同时参数映射还具有很多特性。

首先，像 MyBatis 其他部分一样，参数可以指定许多的数据类型。

```
#{property, javaType = int, jdbcType = NUMERIC}
```

像 MyBatis 的其他部分一样，这个 javaType 是由参数对象决定的，但 HashMap 除外。这个 javaType 应该确保有正确的 typeHandler。

注意：如果传递了一个空值，那这个 jdbc Type 必须接受一个为空的值。

对于需要自定义类型处理的情况，也可以指定一个特殊的 typeHandler 类或者别名，如：

```
#{age, javaType = int, jdbcType = NUMERIC, typeHandler = MyTypeHandler}
```

当然，这看起来更加复杂了，不过，这种情况比较少见。

对于数据类型，可以使用 numericScale 来指定小数位的长度。

```
#{height, javaType = double, jdbcType = NUMERIC, numericScale = 2}
```

最后，mode 属性允许指定 IN、OUT 或 INOUT 参数。如果参数是 OUT 或 INOUT，参数对象属性的实际值将会改变，正如您希望调用一个输出参数，如果 mode = OUT(或者 INOUT)，并且 jdbcType = CURSOR(如 Oracle 的 REFCURSOR)，您必须指定一个 resultMap 映射结果集给这个参数类型。注意这里的 javaType 类型是可选的，如果为空值而 jdbcType = CURSOR 的话，则会自动地将其设给 ResultSet。

```
#{department,
    mode = OUT,
    jdbcType = CURSOR,
    javaType = ResultSet,
    resultMap = departmentResultMap
}
```

MyBatis 也支持高级的数据类型，但当把 mode 设置为 out 的时候，必须把类型名告诉执行语句。例如：

```
#{middleInitial,
    mode = OUT,
    jdbcType = STRUCT,
```

```
    jdbcTypeName = MY_TYPE,
    resultMap = departmentResultMap
}
```

尽管有这些强大的选项，但是大多数情况下只需指定属性名，MyBatis 就会识别其他部分的设置。最多就是给可以为 null 值的列指定 jdbcType：

```
#{firstName}
#{middleInitial, jdbcType = VARCHAR}
#{lastName}
```

默认的情况下，使用#{}语法会促使 MyBatis 生成 PreparedStatement 并且安全地设置 PreparedStatement 参数(=?)值。尽管这是安全、快捷并且是经常使用的，但有时候想直接将未更改的字符串代入到 SQL 语句中，比如说，对于 ORDER BY，可以这样使用：ORDER BY ${columnName}，这样 MyBatis 就不会修改这个字符串了。

警告：这种不加修改地接收用户输入并应用到语句的方式，是非常不安全的。这使用户能够进行 SQL 注入，破坏代码。所以，要么这些字段不允许用户输入，要么用户每次输入后都进行检测和规避。

3.2.7 resultMap 结果集映射

resultMap 元素是 MyBatis 中最重要最强大的元素。与使用 JDBC 从结果集获取数据相比，它可以省掉大部分的代码，也可以做一些 JDBC 不支持的事。事实上，要写一个类似于联结映射(Join Mapping)这样复杂的交互代码，可能需要上千行的代码。设计 resultMaps 的目的，就是只使用简单的配置语句而不需要详细地处理结果集映射，对更复杂的语句除了使用一些必需的语句描述以外，就不需要其他的处理了。

为了方便讲解我们将引入一个用户和他所对应的地址关系的例子。

Userinfo 和 Address 的 JavaBean 定义代码如下所示：

```
package com.test.domain;
public class Userinfo {
    private int userid;
    private String username;
    private String password;
    public int getUserid() {
        return userid;
    }
    public void setUserid(int userid) {
        this.userid = userid;
    }
    public String getUsername() {
        return username;
    }
```

```
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
package com.test.domain;
public class Address {
    private int addrid;
    private String city;
    private String street;
    private String zip;
    public int getAddrid() {
        return addrid;
    }
    public void setAddrid(int addrid) {
        this.addrid = addrid;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getStreet() {
        return street;
    }
    public void setStreet(String street) {
        this.street = street;
    }
    public String getZip() {
        return zip;
    }
    public void setZip(String zip) {
        this.zip = zip;
```

```
    }
}
```

有的简单映射语句并没有使用 resultMap，例如：

```
<select id = "selectUserinfoById" parameterType = "int"
        resultType = "hashmap">
        SELECT * FROM Userinfo WHERE userid = #{userid}
</select>
```

像上面的语句，所有结果集将会自动地映射到以列表为 key 的 HasMap(由 resultType 指定)中。虽然这在许多场合下有用，但是 HashMap 却不是非常好的域模型。更多的情况是使用 JavaBeans 或者 POJOs 作为域模型。MyBatis 支持这两种域模型。考虑上面的 Userinfo 的 JavaBeans 模型。

基于 JavaBeans 规范，上面的类有三个属性：userid、username 和 password。这三个属性对应 select 语句的列名。这样的 JavaBean 可以像 HashMap 一样简单地映射到 ResultSet 结果集。

```
<select id = "selectUserinfoById" parameterType = "int"
        resultType = "userinfo">
        SELECT userid, username, password FROM userinfo WHERE userid = #{userid}
</select>
```

这种情况下，MyBatis 在后台自动生成 resultMap，将列名映射到 JavaBean 的相应属性。如果列名与属性名不匹配，可以使用 select 语法(标准的 SQL 特性)中的将列名取一个别名的方式来进行匹配。代码如下所示：

```
<select id = "selectUserinfoById" parameterType = "int"
        resultType = "userinfo">
    SELECT
    user_id as "userid" ,
    user_name as "username" ,
    password
    FROM userinfo
    WHERE userid = #{userid}
</select>
```

在了解 resultMap 的相关知识之后，看下面的配置代码，这可作为另一种解决列名不匹配的方法：

```
<resultMap id = "userinfoResultMap" type = "com.test.domain.Userinfo">
        <id property = "userid" column = "user_id" />
        <result property = "username" column = "user_name" />
        <result property = "password" column = "password" />
</resultMap>
```

这个语句将会被 resultMap 属性引用(注意，我们没有使用 resultType)，代码如下所示：

```
<select id = "selectUserinfoById" parameterType = "int" resultMap = "userinfoResultMap">
    SELECT userid, username, password FROM userinfo WHERE userid = #{userid}
</select>
```

id、result 元素说明：

```
<id property = "userid" column = "user_id" />
<result property = "username" column = "user_name" />
```

这是最基本的结果集映射。id 和 result 将列映射到属性或简单的数据类型字段(String、int、double、Date 等)。这两者唯一不同的是，在比较对象实例时 id 作为结果集的标识属性。这有助于提高总体性能，特别是应用缓存和嵌套结果映射的时候。

详细的 id、result 元素属性含义如表 3-4 所示。

表 3-4　id、result 元素属性含义

属　性	描　述
property	JavaBean 里需要映射到数据库列的字段或属性。如果 JavaBean 里的属性与给定的名称匹配，就会使用匹配的名字。否则，MyBatis 将搜索给定名称的字段。两种情况下都可以使用逗点加属性形式访问。比如，可以映射到“username”，也可以映射到“address.street.number”
column	数据库里表的列名或者表的列标签别名。与传递给 resultSet.getString(columnName) 的参数名称相同
javaType	完整 Java 类名或别名。如果映射到一个 JavaBean，则 MyBatis 通常会自行检测到。但是，如果要映射到一个 HashMap，那就应该指定 javaType 来限定其类型
jdbcType	表 3-5 将列出 JDBC 的类型。这个属性只在 insert、update 或 delete 的时候针对允许空的列有用。JDBC 需要这个选项，但 MyBatis 不需要。如果直接编写 JDBC 代码，在允许为空值的情况下需要指定这个类型
typeHandler	使用这个属性可以重写默认类型处理器。它的值可以是一个 typeHandler 实现的完整类名，也可以是一个类型别名

MyBatis 在映射字段时会自动调整 JDBC 类型和 Java 类型间的关系，MyBatis 支持的 JDBC 类型如表 3-5 所示。

表 3-5　MyBatis 支持的 JDBC 类型

BIT	FLOAT	CHAR	TIMESTAMP	OTHER	UNDEFINED
TINYINT	REAL	VARCHAR	BINARY	BLOB	NVARCHAR
SMALLINT	DOUBLE	LONGVARCHAR	VARBINARY	CLOB	NCHAR
INTEGER	NUMERIC	DATE	LONGVARBINARY	BOOLEAN	NCLOB
BIGINT	DECIMAL	TIME	NULL	CURSOR	

resultMap 除了上面常用标签外还提供了实例化注入 constructor 元素：

```
<constructor>
    <idArg column = "id" javaType = "int"/>
    <arg column = " username" javaType = " String" />
</constructor>
```

当属性与 DTO 或者与你自己的域模型一起工作的时候，许多场合要用到不变类。通常，包含引用或者查找的数据很少，又或者数据不会改变的表，适合映射到不变类中。构造器注入允许在类实例化后给类设值，这不需要通过 public 方法。MyBatis 同样也支持 private 属性和 JavaBeans 的私有属性达到这一点，但是一些用户可能更喜欢使用构造器注入。构造器元素可以做到这一点，通过下面构造器代码即可实现：

```
public class Userinfo {
    public Userinfo(int userid, String username) {
    }
}
```

为了将结果注入构造器，MyBatis 需要使用它的参数类型来标记构造器。Java 没有办法通过参数名称来反射获得。因此当创建 constructor 元素时，应确保参数是按顺序的并且指定了正确的类型。其他的属性与规则与 id、result 元素的一样，这里不再阐述。

有时候一条数据库查询可能会返回包括各种不同的数据类型的结果集。Discriminator(识别器)元素被设计来处理这种情况以及其他像类继承层次的情况。识别器非常好理解，它就像 Java 里的 switch 语句，discriminator 定义要指定 column 和 javaType 属性。列是 MyBatis 将要取出进行比较的值，javaType 用来确定适当的测试是否正确运行(即使是 String 在大部分情况下也可以工作)。映射代码如下：

```
<resultMap id = "vehicleResult" type = "Vehicle">
<id property = " id" column = "id" />
<result property = "vin" column = "vin"/>
<result property = "year" column = "year"/>
<result property = "make" column = "make"/>
<result property = "model" column = "model"/>
<result property = "color" column = "color"/>
<discriminator javaType = "int" column = "vehicle_type">
<case value = "1" resultMap = "carResult"/>
<case value = "2" resultMap = "truckResult"/>
<case value = "3" resultMap = "vanResult"/>
<case value = "4" resultMap = "suvResult"/>
</discriminator>
</resultMap>
```

在上面代码中，MyBatis 从结果集中取出每条记录，然后比较它的 vehicle type 的值。如果匹配任何 discriminator 中的 case，它将使用由 case 指定的 resultMap。

3.3　SQL 高级映射

3.3.1　拓展 resultMap

我们可以从另外一个<resultMap>拓展出一个新的<resultMap>，这样，原先的属性映射可以继承过来，代码如下所示：

```
<resultMap id = "userinfoResultMap" type = "com.test.domain.Userinfo">
        <id property = "userid" column = "userid" />
        <result property = "username" column = "username" />
        <result property = "password" column = "password" />
</resultMap>
    <resultMap id = "userinfoAndAddressResultMap" type = "com.test.domain.Userinfo" extends =
                "userinfoResultMap">
        <id property = "address.addrid" column = "addrid" />
        <result property = "address.city" column = "city" />
        <result property = "address.street" column = "street" />
        <result property = "address.zip" column = "zip" />
    </resultMap>
```

id 为 userinfoAndAddressResultMap 的 resultMap 拓展了 id 为 userinfoResultMap 的 resultMap。如果只想映射 Userinfo 数据，可以使用 id 为 userinfoResultMap 的 resultMap，配置代码如下所示：

```
<select id = "selectUserinfoById" parameterType = "int" resultMap = "userinfoResultMap">
        SELECT userid, username, password FROM userinfo WHERE userid = #{userid}
</select>
```

如果你想映射 Userinfo 数据和 Address 数据，可以使用 id 为 userinfoAndAddress ResultMap 的 resultMap，配置代码如下所示：

```
<select id = "selectUserinfoAndAddress" parameterType = "int" resultMap = "userinfoAndAddressResultMap">
        SELECT   userid, username, password, a.addrid, city, street, zip
        FROM userinfo u LEFT OUTER JOIN address a ON u.addrid = a.addrid WHERE userid = #{userid}

</select>
```

3.3.2　一对一映射

在我们的域模型样例中，假设每一个用户都有一个与之关联的地址信息。表 userinfo 有一个 addrid 列，是 address 表的外键。

userinfo 表的样例数据如图 3-1 所示。

userid	username	password	addrid
1	test	aaa	1
2	bay	def	2

图 3-1　userinfo 表样例数据

address 表的样例数据，如图 3-2 所示。

addrid	city	street	zip
1	beijing	dongcheng	400056
2	shanghai	huangpu	400023

图 3-2　address 表样例数据

下面让我们看一下怎样获取 userinfo 明细和 address 明细。

Userinfo 和 Address 的 JavaBean 以及映射器 mapper XML 文件定义代码如下所示：

```
package com.test.domain;
public class Userinfo {
    private int userid;
    private String username;
    private String password;
    private Address address;
    public int getUserid() {
        return userid;
    }
    public void setUserid(int userid) {
        this.userid = userid;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
```

```
    public void setPassword(String password) {
        this.password = password;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        this.address = address;
    }
}
package com.test.domain;
public class Address {
    private int addrid;
    private String city;
    private String street;
    private String zip;
    public int getAddrid() {
        return addrid;
    }
    public void setAddrid(int addrid) {
        this.addrid = addrid;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getStreet() {
        return street;
    }
    public void setStreet(String street) {
        this.street = street;
    }
    public String getZip() {
        return zip;
    }
    public void setZip(String zip) {
```

```
        this.zip = zip;
    }
}
```

映射器 mapper XML 代码如下：

```
    <resultMap id = "userinfoAndAddressResultMap" type = "com.test.domain.Userinfo">
        <id property = "userid" column = "userid" />
        <result property = "username" column = "username" />
        <result property = "password" column = "password" />
        <result property = "address.addrid" column = "addrid" />
        <result property = "address.city" column = "city" />
        <result property = "address.street" column = "street" />
        <result property = "address.zip" column = "zip" />
    </resultMap>
    <select id = "selectUserinfoAndAddress" parameterType = "int" resultMap =
            "userinfoAndAddressResultMap">
        SELECT  userid, username, password, a.addrid, city, street, zip
        FROM userinfo u LEFT OUTER JOIN address a ON u.addrid = a.addrid WHERE userid = #{userid}

    </select>
```

我们可以使用圆点记法为内嵌对象的属性赋值。在上述的 resultMap 中，Userinfo 的 address 属性使用了圆点记法被赋上了 address 对应列的值。同样的，我们可以访问任意深度的内嵌对象的属性。

调用方式代码如下：

```
List<Userinfo> list = sqlSession.selectList("com.test.mapper.UserinfoMapper.selectUserinfoAndAddress", 1);
            for (Userinfo info : list) {
                System.out.println(info.getUsername());
                System.out.println(info.getAddress().getCity());
            }
```

上述样例展示了一对一关联映射的一种方法。然而，使用这种方式映射，如果 address 结果需要在其他的 select 映射语句中映射成 Address 对象，我们需要为每一个语句重复这种映射关系。MyBatis 提供了更好地实现一对一关联映射的方法：嵌套结果 resultMap 和嵌套 select 查询语句。接下来，我们将讨论这两种方式。

1) 嵌套 resultMap 实现一对一关系映射

我们可以使用一个嵌套结果 resultMap 方式来获取 Userinfo 及其 Address 信息，代码如下所示：

```
<resultMap id = "addressResultMap" type = "com.test.domain.Address" >
        <id property = "addrid" column = "addrid" />
        <result property = "city" column = "city" />
```

```
        <result property = "street" column = "street" />
        <result property = "zip" column = "zip" />
</resultMap>
    <resultMap id = "userinfoAndAddressResultMap" type = "com.test.domain.Userinfo">
      <id property = "userid" column = "userid" />
        <result property = "username" column = "username" />
        <result property = "password" column = "password" />
        <association property = "address" resultMap = "addressResultMap"></association>
    </resultMap>
<select id = "selectUserinfoAndAddress" parameterType = "int" resultMap = "userinfoAndAddressResultMap">
        SELECT   userid, username, password, a.addrid, city, street, zip
        FROM userinfo u LEFT OUTER JOIN address a ON u.addrid = a.addrid WHERE userid = #{userid}

</select>
```

association 元素处理“has-one”(一对一)这种类型关系。在上述的代码中，我们使用<association>元素引用了另外的在同一个 XML 文件中定义的<resultMap> addressResultMap，这里的 addressResultMap 结果集映射可以重复使用；如果不需要重复，则可以直接在 XML 里嵌套这个联合查询的映射结果。代码如下：

```
<resultMap id = "userinfoAndAddressResultMap" type = "com.test.domain.Userinfo">
      <id property = "userid" column = "userid" />
        <result property = "username" column = "username" />
        <result property = "password" column = "password" />
        <association property = "address" column = "addrid" javaType = "com.test.domain.Address">
        <id property = "addrid" column = "addrid" />
        <result property = "city" column = "city" />
        <result property = "street" column = "street" />
        <result property = "zip" column = "zip" />
        </association>
    </resultMap>
```

<association>元素和其他的集映射工作方式差不多，指定 property、column、javaType(通常 MyBatis 会自动识别)、jdbcType(如果需要)、typeHandler。不同的地方是需要告诉 MyBatis 如何加载一个联合查询。MyBatis 使用两种方式来加载：

results：通过嵌套映射结果(nested result mappings)来处理连接结果集(joined results)的重复子集。

select：通过执行另一个返回复杂类型的映射 SQL 语句(即引用外部定义好的 SQL 语句块)。

首先，检查一下元素属性。正如我们看到的，它不同于只有 resultMap 和 select 所示属性的结果映射，如表 3-6 所示。

表 3-6 association 属性含义

属 性	描 述
property	映射数据库列的字段或属性。如果 JavaBean 的属性与给定的名称匹配，就会使用匹配的名字。否则，MyBatis 将搜索给定名称的字段。两种情况下都可以使用逗点的属性形式。比如，可以映射到“username”，也可以映射到更复杂点的“address.street.number”
column	数据库的列名或者列标签别名。与传递给 resultSet.getString(columnName)的参数名称相同。 **注意**：在处理组合键时，可以使用 column = “{prop1 = col1, prop2 = col2}” 这样的语法，设置多个列名传入到嵌套查询语句。这就会把 prop1 和 prop2 设置到目标嵌套选择语句的参数对象中
javaType	完整 Java 类名或别名(参考上面的内置别名列表)。如果映射到一个 JavaBean，那 MyBatis 通常会自行检测到。然而，如果映射到一个 HashMap，则应该明确指定 javaType 来确保所需行为
jdbcType	支持 JDBC 类型列表中列出的 JDBC 类型。这个属性只在 insert，update 或 delete 的时候针对允许空的列有用。JDBC 需要这项，但 MyBatis 不需要。如果直接编写 JDBC 代码，在允许为空值的情况下需要指定这个类型
typeHandler	使用这个属性可以重写默认类型处理器。它的值可以是一个 typeHandler 实现的完整类名，也可以是一个类型别名
联合嵌套结果集(Nested Results for Association)	
resultMap	一个可以映射联合嵌套结果集，这是用替代的方式去调用另一个查询语句。它允许您去联合多个表到一个结果集里。这样的结果集可能包括冗余的、重复的需要分解和正确映射到一个嵌套对象视图的数据组。简言之，MyBatis 把结果映射链接到一起，用来处理嵌套
联合嵌套选择(Nested Select for Association)	
select	通过这个属性，通过 ID 引用另一个加载复杂类型的映射语句。从指定列属性中返回的值，将作为参数设置给目标 select 语句。下文将会举例说明。 **注意**：在处理组合键时，可以使用 column = “{prop1 = col1, prop2 = col2}” 这样的语法，设置多个列名传入到嵌套语句。这就会把 prop1 和 prop2 设置到目标嵌套语句的参数对象中

2) 嵌套引入查询实现一对一关系映射

讨论完嵌套 resultMap 实现一对一关系映射后，接下来我们将学习嵌套引入查询实现一对一关系映射。这个方法虽然简单，但是对于大数据集或列表查询，就不尽如人意了。这个问题被称为“N+1”选择问题。我们可以使用如下代码来获取用户 Userinfo 的地址 Address：

```
<resultMap id = "addressResultMap" type = "com.test.domain.Address">
        <id property = "addrid" column = "addrid" />
        <result property = "city" column = "city" />
        <result property = "street" column = "street" />
        <result property = "zip" column = "zip" />
    </resultMap>
<select id = "selectAddressById" parameterType = "int" resultMap = "addressResultMap">
        SELECT * FROM address WHERE addrid = #{addrid}
    </select>
<resultMap id = "userinfoAndAddressResultMap" type = "com.test.domain.Userinfo">
        <id property = "userid" column = "userid" />
        <result property = "username" column = "username" />
        <result property = "password" column = "password" />
        <association property = "address" column = "addrid" select = "selectAddressById" />
</resultMap>
<select id = "selectUserinfoAndAddress" parameterType = "int"
        resultMap = "userinfoAndAddressResultMap">
        SELECT * FROM userinfo where userid = #{userid}
</select>
```

在此方式中，<association>元素的 select 属性被设置成 id 为 selectAddressById 的语句。这里，两个分开的 SQL 语句将会在数据库中执行，第一个调用 selectUserinfoAndAddress 加载 userinfo 信息，而第二个调用 findAddressById 加载 address 信息。Addr_id 列的值将被作为输入参数传递给 selectAddressById 语句。

上述提到的“N+1”问题的产生过程如下：

(1) 执行单条 SQL 语句去获取一个列表的记录(“+1”上面例子中的 selectUserinfoAndAddress 加载 userinfo 信息)

(2) 对列表中的每一条记录，再执行一个联合 select 语句来加载每条记录更加详细的信息(“N”上面例子中的 findAddressById 加载 address 信息)。

因此上面例子中执行一条 SQL 语句获取到 10 条 userinfo，这 10 条 userinfo 记录的每一条再执行一条 SQL 语句获取 address 信息，所以总共会执行 11 次查询，当数据量比较大时，这种方式并非理想之选。

我们在上面的例子中已经看到如何处理“一对一”(“has one”)类型的联合查询。但是对于“一对多”(“has many”)的情况如何处理呢？这个问题在下一节讨论。

3.3.3 一对多映射

在一对多映射中，我们的域模型是部门和员工之间的关系，一个部门可以有一个或者多个员工，这意味着部门和员工之间存在一对多的映射关系。我们可以使用<collection>元素将一对多类型的结果映射到一个对象集合上。

dempartment 表的样例数据如图 3-3 所示。

	depid	depname
	1	Business Office
	2	Human Resources Department
*	3	Sales Department

图 3-3　dempartment 表样例数据

employee 表的样例数据如图 3-4 所示。

	empid	empname	depid
	1	Bill	1
	2	John	2
▶	3	Tigger	2
	4	bay	3

图 3-4　employee 表样例数据

在上述表中，Business Office 有两名员工，Human Resources Department 有两名员工，Sales Department 有一名员工。

Employee 域对象代码如下：

```
package com.test.domain;
public class Employee {
    private int empid;
    private String empname;
    public int getEmpid() {
        return empid;
    }
    public void setEmpid(int empid) {
        this.empid = empid;
    }
    public String getEmpname() {
        return empname;
    }
    public void setEmpname(String empname) {
        this.empname = empname;
    }
}
```

Department 域对象代码如下：

```
package com.test.domain;
import java.util.List;
```

```
public class Department {
    private int depid;
    private String depname;
    private List<Employee> employees;
    public int getDepid() {
        return depid;
    }
    public void setDepid(int depid) {
        this.depid = depid;
    }
    public String getDepname() {
        return depname;
    }
    public void setDepname(String depname) {
        this.depname = depname;
    }
    public List<Employee> getEmployees() {
        return employees;
    }
    public void setEmployees(List<Employee> employees) {
        this.employees = employees;
    }
}
```

上面一对多关系中，如果要获取部门信息及部门所对应的员工信息，MyBatis 提供了<collection>元素被用来将多行员工结果映射成一个过程 Employee 对象的集合。和一对一映射一样，我们可以使用嵌套结果 resultMap 和嵌套 select 语句两种方式映射实现一对多映射。

1) 使用内嵌结果 resultMap 实现一对多映射

要得到部门及部门所对应的员工信息可以使用嵌套结果 resultMap 的方式，代码如下：

```
<resultMap id = "employeeResultMap" type = "com.test.domain.Employee">
        <id property = "empid" column = "empid" />
        <result property = "empname" column = "empname" />
</resultMap>
    <resultMap id = "departmentResultMap" type = "com.test.domain.Department">
        <id property = "depid" column = "depid" />
        <result property = "depname" column = "depname" />
        <collection property = "employees" resultMap = "employeeResultMap"></collection>
    </resultMap>
    <select id = "selectDepartmentAndEmployeeById" parameterType = "int"
```

```
        resultMap = "departmentResultMap">
        SELECT dep.depid, dep.depname, emp.empid, emp.empname
        FROM
        department dep LEFT OUTER JOIN employee emp ON dep.depid = emp.depid
        WHERE
        dep.depid = #{depid}
    </select>
```

这里我们简单地使用了一个 JOINS 连接的 select 语句获取部门及其员工信息。

<collection>元素的 resultMap 属性设置成 employeeResultMap，employeeResultMap 包含 Employee 对象属性与表列名之间的映射。

2) 使用嵌套 select 语句实现一对多映射

要使用嵌套 select 方式得到上面的结果，代码如下：

```
<resultMap id = "employeeResultMap" type = "com.test.domain.Employee">
        <id property = "empid" column = "empid" />
        <result property = "empname" column = "empname" />
</resultMap>
<resultMap id = "departmentResultMap" type = "com.test.domain.Department">
        <id property = "depid" column = "depid" />
        <result property = "depname" column = "depname" />
        <collection property = "employees" select = "selectEmployeeById" column = "depid"></collection>
</resultMap>
<select id = "selectEmployeeById" parameterType = "int"
        resultMap = "employeeResultMap">
        SELECT emp.empid, emp.empname
        FROM
        employee emp
        WHERE
        emp.empid = #{empid}
</select>
<select id = "selectDepartmentAndEmployeeById" parameterType = "int"
        resultMap = "departmentResultMap">
        SELECT dep.depid, dep.depname
        FROM
        department dep
        WHERE
        dep.depid = #{depid}
</select>
```

在这种方式中，<aossication>元素的 select 属性被设置成 id 为 selectEmployeeById 的语句，用来触发单独的 SQL 查询加载员工信息。depid 这一列值将会作为输入参数传递给 selectEmployeeById 语句。一对多嵌套 select 语句和一对一嵌套 select 语句的实现方式一样，也存在“N+1”的性能问题。对于较多数据的场合也不是一个很好的选择。

3.3.4　cache 和 cache-ref 元素

1) cache 元素

MyBatis 包含一个强大的、可配置并可定制的查询缓存机制。MyBatis 3 的缓存实现有了许多改进，使其更强大更容易配置。默认的情况下，缓存是没有开启的，除了会话缓存以外。会话缓存可以提高性能，且能解决循环依赖。开启二级缓存，只需要在 SQL 映射文件中加入简单的一行：

```
<cache/>
```

这句简单的语句作用如下：

- 所有映射文件里的 select 语句的结果都会被缓存。
- 所有映射文件里的 insert、update 和 delete 语句执行都会清空缓存。
- 缓存使用最近最少使用算法(LRU)来回收。
- 缓存不会被设定的时间所清空。
- 每个缓存可以存储 1024 个列表或对象的引用(不管查询方法返回的是什么)。
- 缓存将作为“读/写”缓存，意味着检索的对象不是共享的且可以被调用者安全地修改，而不会被其他调用者或者线程干扰。

所有这些特性都可以通过 cache 元素进行修改。代码如下所示：

```
<cache
eviction = "FIFO"
flushInterval = "60000"
size = "512"
readOnly = "true"/>
```

这种高级的配置创建一个每 60 秒刷新一次的 FIFO 缓存，存储 512 个结果对象或列表的引用，并且返回的对象是只读的。因此在不用的线程里的调用者修改它们可能会引起冲突。

可用的回收算法如下：

- LRU (最近最少使用)：移出最近最长时间内都没有被使用的对象。
- FIFO (先进先出)：移除最先进入缓存的对象。
- SOFT (软引用)：基于垃圾回收机制和软引用规则来移除对象(空间内存不足时才进行回收)。
- WEAK (弱引用)：基于垃圾回收机制和弱引用规则来移除对象(垃圾回收器扫描到时即进行回收)。
- flushInterval：可设置为任何正整数，代表一个以毫秒为单位的合理时间。默认是没有设置，因此没有刷新间隔时间被使用，在语句每次调用时才进行刷新。

• Size：可以设置为一个正整数，需要留意要缓存对象的大小和环境中可用的内存空间，默认是 1024。

• readOnly：可以被设置为 true 或 false。只读缓存将对所有调用者返回同一个实例。因此这些对象都不能被修改，这可以极大地提高性能。可写的缓存将通过序列化来返回一个缓存对象的拷贝。这会比较慢，但是比较安全。所以默认值是 false。

一个缓存配置和缓存实例都绑定到一个 sql Map 文件命名空间。因此，所有相同命名空间的语句都绑定相同的缓存。配置语句可以修改如何与这个缓存相匹配，或者使用两个简单的属性来完全排除它们自己。默认情况下，配置语句如下所示：

```
<select ... flushCache = " false" useCache = " true"/>
<insert ... flushCache = " true" />
<update ... flushCache = " true" />
<delete ... flushCache = " true" />
```

因为有默认值，所以不需要使用这种方式明确地配置这些语句。如果想改变默认的动作，只需要设置 flushCache 和 useCache 属性即可。例如，对一个 select 语句不使用缓存，可以设置 useCache = “false”。

除了内建的缓存支持，MyBatis 也提供了与第三方缓存类库如 Ehcache、OSCache、Hazelcast 的集成支持。你可以在 MyBatis 官方网站 https://code.google.com/p/mybatis 上找到关于继承第三方缓存类库的更多信息。

2) cache-ref 元素

上面讨论在某一个命名空间里使用<cache>元素配置或者刷新缓存。但有可能要在不同的命名空间里共享同一个缓存配置或者实例。在这种情况下，则可以使用 cache-ref 元素来引用另外一个缓存。例如下列代码：

```
<cache-ref namespace = "com.test.mapper.DepartmentMapper" />
```

3.4 动态 SQL

MyBatis 最强大的特性之一就是它的动态语句功能。如果您以前使用过 JDBC 或者类似框架，就会明白把 SQL 语句条件连接在一起是很繁琐的，要确保不能忘记空格或者不要在 columns 列后面省略一个逗号等。动态语句能够完全解决这些问题。

尽管与动态 SQL 一起工作不是在开一个 party，但是 MyBatis 确实能通过在任何映射 SQL 语句使用强大的动态 SQL 来改进这些状况。

动态 SQL 元素对于任何使用过 JSTL 或者类似于 XML 之类的文本处理器的人来说，都是非常熟悉的。在 MyBatis 3 版中有了许多的改进，现在只剩下差不多二分之一的元素。MyBatis 使用基于强大的 OGNL 表达式来消除大部分元素。MyBatis 通过使用 if、choose(when, otherwise)、trim(where, set)和 foreach 等元素提供对构造动态 SQL 语句的高级别支持，接下来，我们将依次介绍其使用方法。

3.4.1　if 元素

if 就是简单的条件判断，利用 if 语句我们可以实现某些简单的条件选择，实现条件查询用户信息，代码如下所示：

```
<select id = "selectUserinfo" parameterType = "userinfo" resultType = "userinfo">
        SELECT * FROM userinfo WHERE 1 = 1
        <if test = "username != null">
            AND username like #{username}
        </if>
        <if test = "nickname != null">
            AND nickname like #{nickname}
        </if>
    </select>
```

如果你提供了 username 参数，那么就要满足 username like #{username}；如果你也提供了 nickname 参数，那么就要满足 nickname like #{ nickname }；之后就是返回满足这些条件的所有 userinfo，这是非常有用的一个功能。以往使用其他类型框架或者直接使用 JDBC 的时候，如果要达到同样的选择效果，就需要拼 SQL 语句，这是极其麻烦的，而上述的动态 SQL 语句就要简单多了。

3.4.2　choose、when、otherwise 元素

有时候我们不想应用所有的条件，而是想从多个选项中选择一个。与 Java 中的 switch 语句相似，MyBatis 提供了一个 choose 元素。代码如下所示：

```
<select id = "selectUserinfo1" parameterType = "userinfo" resultType = "userinfo">
        SELECT * FROM userinfo WHERE 1 = 1
        <choose>
        <when test = "username != null">
            AND username like #{username}
        </when>
        <when test = "nickname != null">
            AND nickname like #{nickname}
        </when>
        <otherwise>
         AND nickname = "bay"
        </otherwise>
        </choose>
    </select>
```

when 元素表示当 when 中的条件满足的时候就输出其中的内容，跟 Java 中的 switch 效

果差不多的是按照条件的顺序，当 when 中有条件满足的时候，就会跳出 choose，即所有的 when 和 otherwise 条件中，只有一个会输出，当所有的条件都不满足的时候就输出 otherwise 中的内容。所以上述语句的意思非常简单，当 username != null 的时候就输出 AND username like #{username}，不再往下判断条件；当 username 为空且 nickname != null 的时候就输出 AND nickname like #{nickname}；当所有条件都不满足的时候就输出 otherwise 中的内容 AND nickname = "bay"。

3.4.3 where、trim、set 元素

1) where 元素

在 3.4.2 的代码中我们在 WHERE 后面带上一个“1 = 1”，为什么我们要把“1 = 1”带上呢？假如我们把 WHERE 1 = 1 去掉，如下代码所示：

```
<select id = "selectUserinfo" parameterType = "userinfo" resultType = "userinfo">
        SELECT * FROM userinfo
        <if test = "username != null">
            AND username like #{username}
        </if>
        <if test = "nickname != null">
            AND nickname like #{nickname}
        </if>
    </select>
```

假如我们这里传入的 userinfo 里 username 和 nickname 都为空时语句正常执行，但若 username 和 nickname 其中之一不为空，这时语句就有问题了，即使再把 AND 去掉，把 where 加上依然不能解决问题，当在后面加上一个 WHERE 1 = 1，问题便得到解决。在 MyBatis 中 where 元素能智能地处理 SQL 的 where 条件。上面的代码可修改成如下代码：

```
<select id = "selectUserinfo" parameterType = "userinfo" resultType = "userinfo">
        SELECT * FROM userinfo
        <where>
            <if test = "username != null">
                AND username like #{username}
            </if>
            <if test = "nickname != null">
                AND nickname like #{nickname}
            </if>
        </where>
    </select>
```

上面 where 元素的作用是会在写入 where 元素的地方输出一个 where，另外一个优点是不需要考虑 where 元素里面的条件输出是什么，MyBatis 会智能地处理，如果所有的条件都

不满足，那么 MyBatis 就会查出所有的记录，如果输出后是 and 开头的，MyBatis 会把第一个 and 忽略，如果是 or 开头的，MyBatis 也会把它忽略；此外，在 where 元素中不需要考虑空格的问题，MyBatis 会智能地加上。像上述例子中，如果 username = null，而 nickname != null，那么输出的整个语句会是 SELECT * FROM userinfo WHERE nickname like #{nickname}，而不是 SELECT * FROM userinfo WHERE AND nickname like #{nickname}，因为 MyBatis 会智能地把首个 and 或 or 忽略掉。

2) trim 元素

trim 元素的主要功能是可以在自己包含的内容前加上某些前缀，也可以在其后加上某些后缀，与之对应的属性是 prefix 和 suffix；可以把包含内容的首部某些内容覆盖，即忽略，也可以把尾部的某些内容覆盖，对应的属性是 prefixOverrides 和 suffixOverrides。正因为 trim 有这样的功能，所以我们可以非常简单地利用 trim 来代替 where 元素的功能，如果 where 元素的行为并没有完全按您想象的那样，您还可以使用 trim 元素来自定义。

上面的条件查询用户信息代码可改成：

```
<select id = "selectUserinfo" parameterType = "userinfo"
        resultType = "userinfo">
        SELECT * FROM userinfo
        <trim prefix = "WHERE" prefixOverrides = "AND |OR ">
            <if test = "username != null">
                AND username like #{username}
            </if>
            <if test = "nickname != null">
                AND nickname like #{nickname}
            </if>
        </trim>
    </select>
```

3) set 元素

set 元素主要用在更新操作的时候，它的主要功能和 where 元素其实是差不多的，就是在包含的语句前输出一个 set，如果包含的语句是以逗号结束则把该逗号忽略，如果 set 包含的内容为空则会出错。有了 set 元素就可以动态地更新那些修改了的字段。下面是一段更新操作的代码：

```
<update id = "updateUserinfo" parameterType = "userinfo">
        UPDATE userinfo
        <set>
            <if test = "username != null">
                username = #{username}
            </if>
            <if test = "nickname != null">
                nickname = #{nickname}
```

```
            </if>
        </set>
        WHERE userid = #{userid}
    </update>
```

注意：上述代码中，如果 set 中一个条件都不满足，即 set 中包含的内容为空的时候就会报错。

3.4.4 foreach 元素

foreach 主要用在构建 in 条件中，它可以在 SQL 语句中迭代一个集合。foreach 元素的属性主要有 item、index、collection、open、separator、close。item 表示集合中每一个元素进行迭代时的别名；index 指定一个名字，表示在迭代过程中，每次迭代到的位置；open 表示该语句以什么开始；separator 表示在每次进行迭代之间以什么符号作为分隔符；close 表示以什么结束。在使用 foreach 的时候最关键的也是最容易出错的就是 collection 属性，该属性是必须指定的，但是在不同情况下，该属性的值是不一样的，主要有以下三种情况：

(1) 如果传入的是单参数且参数类型是一个 list 的时候，collection 属性值为 list；

(2) 如果传入的是单参数且参数类型是一个 array 数组的时候，collection 的属性值为 array；

(3) 如果传入的参数是多个的时候，我们就需要把它们封装成一个 Map 了。当然单参数也可以封装成 Map，实际上如果在传入参数的时候，在 MyBatis 里面也是会把它封装成一个 Map 的。Map 的 key 就是参数名，所以这个时候 collection 属性值就是传入的 list 或 array 对象在自己封装的 Map 里面的 key。

(1) 传入单参数 list 的类型映射配置代码如下：

```
<select id = "selectUserinfo" resultType = "userinfo">
        SELECT * FROM userinfo where userid in
         <foreach collection = "list" index = "index" item = "item" open = "(" separator=", " close=")">
    #{item}
    </foreach>
</select>
```

调用时传入参数的 Java 代码如下：

```
List<Integer> ids = new ArrayList<Integer>();
            ids.add(1);
            ids.add(2);
            ids.add(3);
```

(2) 传入单参数 array 数组的类型映射配置代码如下：

```
<select id = "selectUserinfo5" resultType = "userinfo">
        SELECT * FROM userinfo where userid in
        <foreach collection = "array" index = "index" item = "item" open = "("
```

```
            separator = ", " close = ")">
            #{item}
        </foreach>
    </select>
```

调用时传入参数的代码如下：

```
int [] ids = new int[]{1, 2, 3};
```

(3) 传入参数封装成 Map 的多参数类型映射配置代码如下：

```
<select id = "selectUserinfo6" resultType = "userinfo">
        SELECT * FROM userinfo where username like "%"#{username}"%" and userid in
        <foreach collection = "ids" index = "index" item = "item" open = "("
            separator = ", " close = ")">
            #{item}
        </foreach>
    </select>
```

调用时传入参数的代码如下：

```
int [] ids = new int[]{1, 2, 3};
Map<String, Object> params = new HashMap<String, Object>();
params.put("ids", ids);
params.put("username", "bay");
```

3.5　注解配置 SQL 映射器

在前面的学习中，我们了解了怎样在映射器 mapper XML 配置文件中配置映射语句。MyBatis 也支持使用注解来配置映射语句。当使用基于注解的映射器接口时，我们不再需要在 XML 配置文件中配置。如果愿意，也可以同时使用基于 XML 和基于注解的映射语句。

3.5.1　@Select 查询语句

可以在 mapper 接口方法上使用@Select 注解来定义一个 select 映射语句。定义方式代码如下：

```
package com.test.mapper;
import java.util.List;
import org.apache.ibatis.annotations.Select;
import com.test.domain.Student;
public interface StudentMapper {
    @Select("SELECT * FROM student WHERE stuid = #{stuid}")
    public List<Student> selectStudentByStuId(Integer stuid);
```

```
}
```

使用了@Select 注解的 selectStudentByStuId()方法将返回查询的结果集。

3.5.2 @Insert 插入语句

可以在 Mapper 接口方法上使用@Insert 注解来定义一个 insert 映射语句。代码如下所示：

```
package com.test.mapper;
import java.util.List;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Select;
import com.test.domain.Student;
public interface StudentMapper {
    @Insert("INSERT INTO student(stuid, stuname) VALUES(#{stuid}, #{stuname})")
    public int insertStudent(Student student);
}
```

使用了@Insert 注解的 insertStudent()方法将返回 insert 语句执行后影响的行数。

1) 自动生成主键

对于支持 AUTO_INCREMENT 的数据库，可以使用@Options 注解的 userGeneratedKeys 和 keyProperty 属性让数据库产生 auto_increment(自增长)列的值，然后将生成的值设置到输入参数对象的属性中。

```
package com.test.mapper;
import java.util.List;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Options;
import org.apache.ibatis.annotations.Select;
import com.test.domain.Student;
public interface StudentMapper {
    @Insert("INSERT INTO student(stuname) VALUES(#{stuname})")
    @Options(useGeneratedKeys = true, keyProperty = "stuid")
    public int insertStudent(Student student);
}
```

2) 指定主键

有一些数据库如 Oracle，并不支持 AUTO_INCREMENT 列属性，它使用序列(SEQUENCE)来产生主键的值。我们可以使用@SelectKey 注解来为任意 SQL 语句指定主键值，作为主键列的值。假设我们有一个名为 STU_ID_SEQ 的序列来生成 stuid 主键值，代码如下：

```
package com.test.mapper;
```

```
import java.util.List;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Options;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.SelectKey;
import com.test.domain.Student;
public interface StudentMapper {
    @Insert("INSERT INTO student(stuname) VALUES(#{stuname})")
    @SelectKey(statement = "SELECT STU_ID_SEQ.NEXTVAL FROM DUAL", keyProperty = "stuid",
resultType = int.class, before = true)
    public int insertStudent(Student student);
}
```

这里我们使用了@SelectKey 来生成主键值，并且存储到了 student 对象的 stuid 属性上。由于我们设置了 before = true，该语句将会在执行 insert 语句之前执行。

如果使用序列作为触发器来设置主键值，我们可以在 insert 语句执行后，从 sequence_name.currval 获取数据库产生的主键值。代码如下：

```
package com.test.mapper;
import java.util.List;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Options;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.SelectKey;
import com.test.domain.Student;
public interface StudentMapper {
    @Insert("INSERT INTO student(stuname) VALUES(#{stuname})")
    @SelectKey(statement = "SELECT STU_ID_SEQ.CURRVAL FROM DUAL", keyProperty = "stuid",
resultType = int.class, before = false)
    public int insertStudent(Student student);
}
```

3.5.3　@Update 修改语句

可以在 mapper 接口方法上使用@Update 注解来定义一个 update 映射语句。代码如下所示：

```
package com.test.mapper;
import java.util.List;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Options;
import org.apache.ibatis.annotations.Select;
```

```
import org.apache.ibatis.annotations.Update;
import com.test.domain.Student;
public interface StudentMapper {
    @Update("UPDATE student SET stuname = #{stuname} WHERE stuid = #{stuid}")
    public int updateStudent(Student student);
}
```

使用了@Update 注解的 updateStudent()方法将返回执行了 update 语句后影响的行数。

3.5.4 @Delete 删除语句

可以在 mapper 接口方法上使用@Delete 注解来定义一个 delete 映射语句。代码如下所示：

```
package com.test.mapper;
import java.util.List;
import org.apache.ibatis.annotations.Delete;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Options;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.Update;
import com.test.domain.Student;
public interface StudentMapper {
    @Delete("DELETE FROM student WHERE stuid = #{stuid}")
    public int deleteStudent(Integer stuid);
}
```

使用了@Delete 注解的 deleteStudent()方法将返回执行了 delete 语句后影响的行数。

3.5.5 @ResultMap 结果映射

我们可以将查询结果通过别名或者是@Results 注解与 JavaBean 属性映射起来，用来实现 resultMap 结果集映射的功能，执行 select 查询代码如下：

```
package com.test.mapper;
import java.util.List;
import org.apache.ibatis.annotations.Delete;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Options;
import org.apache.ibatis.annotations.Result;
import org.apache.ibatis.annotations.Results;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.Update;
```

```
import com.test.domain.Student;
public interface StudentMapper {
    @Select("SELECT * FROM student WHERE stuid = #{stuid}")
    @Results({
        @Result(id = true, column = "stuid", property = "stuid"),
        @Result(column = "stuname", property = "stuname")
    })
    public List<Student> selectStudentByStuId(Integer stuid);
}
```

如果 results 需要重复利用，我们可以创建一个映射器 mapper 配置文件，配置<resultMap>元素，然后使用@ResultMap 注解引用此<resultMap>。

在 StudentMapper.xml 中定义一个 ID 为 StudentResultMap 的<resultMap>，代码如下所示：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace = "com.test.mapper.StudentMapper">
    <resultMap id = "studentResultMap" type = "com.test.domain.Student">
        <id property = "stuid" column = "stuid" />
        <result property = "stuname" column = "stuname" />
    </resultMap>
</mapper>
```

在 StudentMapper.java 中，可以使用@ResultMap 注解来引用 XML 里名为 studentResultMap 的 resultMap 结果映射，代码如下：

```
package com.test.mapper;
import java.util.List;
import org.apache.ibatis.annotations.Delete;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Options;
import org.apache.ibatis.annotations.ResultMap;
import org.apache.ibatis.annotations.Select;
import org.apache.ibatis.annotations.Update;
import com.test.domain.Student;
public interface StudentMapper {
    @Select("SELECT * FROM student WHERE stuid = #{stuid}")
    @ResultMap("com.test.mapper.StudentMapper.studentResultMap")
    public List<Student> selectStudentByStuId(Integer stuid);
```

```
}
```

3.5.6 @One 一对一映射

MyBatis 提供了@One 注解来使用嵌套 select 语句(Nested-Select)加载一对一关联查询数据。以下是获取 userinfo 和 address 信息的代码示例。

mapper 接口注解：

```
package com.test.mapper;
import java.util.List;
import org.apache.ibatis.annotations.One;
import org.apache.ibatis.annotations.Result;
import org.apache.ibatis.annotations.Results;
import com.test.domain.Userinfo;
public interface UserinfoMapper {
@Select("SELECT * FROM address WHERE addrid = #{addrid}")
    public Address selectAddressById(int addrid);
    @Results(
            {
            @Result(id = true, column = "userid", property = "userid"),
            @Result(column = "username", property = "username"),
            @Result(column = "password", property = "password"),
            @Result(column = "nickname", property = "nickname"),
            @Result(property = "address", column = "addrid",
            one = @One(select = "com.test.mapper.UserinfoMapper.selectAddressById"))
            })
    public List<Userinfo> selectUserinfoById(Integer userid);
}
```

这里我们使用了@One 注解的 select 属性来指定一个使用了完全限定名的方法，该方法会返回一个 Address 对象。使用 column = “addrid”，则 userinfo 表中列 addrid 的值将会作为输入参数传递给 selectAddressById()方法。如果@One select 查询返回了多行结果，则会抛出 TooManyResultsException 异常。

除了使用嵌套 select 语句的方式进行一对一的关联查询外，我们还可以在配置文件里配置 resultMap 使用嵌套结果方式 resultMap 加载一对一关联的查询。

resultMap 配置代码如下所示：

```
<resultMap id = "addressResultMap" type = "com.test.domain.Address">
        <id property = "addrid" column = "addrid" />
        <result property = "city" column = "city" />
        <result property = "street" column = "street" />
        <result property = "zip" column = "zip" />
```

```
    </resultMap>
<resultMap id = "userinfoAndAddressResultMap" type = "com.test.domain.Userinfo">
        <id property = "userid" column = "userid" />
        <result property = "username" column = "username" />
        <result property = "password" column = "password" />
        <association property = "address" resultMap = "addressResultMap"></association>
    </resultMap>
```

mapper 接口代码如下所示：

```
package com.test.mapper;
import java.util.List;
import org.apache.ibatis.annotations.ResultMap;
import org.apache.ibatis.annotations.Select;
import com.test.domain.Userinfo;
public interface UserinfoMapper {
    @Select("SELECT userid, username, password, a.addrid, city, street, zip FROM userinfo u LEFT OUTER
JOIN address a ON u.addrid = a.addrid WHERE userid = #{userid}")
    @ResultMap("com.test.mapper.UserinfoMapper.userinfoAndAddressResultMap")
    public List<Userinfo> selectUserinfoById(Integer userid);
}
```

3.5.7　@Many 一对多映射

MyBatis 提供了@Many 注解，用来使用嵌套 Select 语句加载一对多关联查询。以下为查询部门及其员工信息的代码示例：

```
package com.test.mapper;
import java.util.List;
import org.apache.ibatis.annotations.Many;
import org.apache.ibatis.annotations.Result;
import org.apache.ibatis.annotations.Results;
import org.apache.ibatis.annotations.Select;
import com.test.domain.Department;
import com.test.domain.Employee;
public interface DepartmentMapper {
    @Select("SELECT emp.empid, emp.empname FROM employee emp WHERE emp.empid = #{empid}")
    @Results({ @Result(id = true, column = "empid", property = "empid"),
            @Result(column = "empname", property = "empname")
    })
    public List<Employee> selectEmployeesById(int empid);
    @Select("SELECT dep.depid, dep.depname FROM department dep  WHERE   dep.depid = #{depid}")
```

```
    @Results({
            @Result(id = true, column = "depid", property = "depid"),
            @Result(column = "depname", property = "depname"),
            @Result(property = "employees", column = "depid", many = @Many(select =
                    "com.test.mapper.selectEmployeesById")) })
    public Department selectDepartmentAndEmployeeById(int depid);
}
```

这里我们使用了@Many 注解的 select 属性来指向一个完全限定名称的方法，该方法将返回一个 List<Employee>对象。使用 column=“depid”，department 表的 depid 列值将会作为输入参数传递给 selectEmployeesById()方法。除了嵌套 select 语句来关联查询一对多映射外，我们还可以结合在配置文件里配置 resultMap 使用嵌套结果 resultMap 来加载一对多关联的查询。

resultMap 配置代码如下所示：

```
    <resultMap id = "employeeResultMap" type = "com.test.domain.Employee">
        <id property = "empid" column = "empid" />
        <result property = "empname" column = "empname" />
    </resultMap>
    <resultMap id = "departmentResultMap" type = "com.test.domain.Department">
        <id property = "depid" column = "depid" />
        <result property = "depname" column = "depname" />
        <collection property = "employees" resultMap = "employeeResultMap"></collection>
    </resultMap>
```

mapper 接口代码如下所示：

```
package com.test.mapper;
import org.apache.ibatis.annotations.Select;
import com.test.domain.Department;
public interface DepartmentMapper {
    @Select("SELECT dep.depid, dep.depname,emp.empid,emp.empname FROM department dep LEFT
OUTER JOIN employee emp ON dep.depid = emp.depid WHERE dep.depid = #{depid}")
    public Department selectDepartmentAndEmployeeById(int depid);
}
```

3.5.8 @SelectProvider 动态查询语句

有时候我们需要根据输入条件动态地构建 SQL 语句。MyBatis 提供了各种注解如@InsertProvider、@UpdateProvider，@DeleteProvider 和@SelectProvider，来帮助构建动态 SQL 语句，然后让 MyBatis 执行这些 SQL 语句。现在让我们来看一个使用@SelectProvider 注解来创建简单的 select 映射语句的例子。

创建一个 DepartmentProvider.java 类，以及 selectDepartmentByIdSql()方法，代码如下所示：

```
package com.test.provider;
import java.util.Map;
public class DepartmentProvider {
    public String selectDepartmentByIdSql( Map<String, Object> parameters) {
        int   depid = Integer.valueOf(parameters.get("depid").toString());
        return "SELECT dep.depid, dep.depname FROM department dep WHERE dep.depid = " + depid;
    }
}
```

在 DepartmentMapper.java 接口中创建一个映射语句，代码如下所示：

```
package com.test.mapper;
import java.util.List;
import org.apache.ibatis.annotations.Param;
import org.apache.ibatis.annotations.SelectProvider;
import com.test.domain.Department;
import com.test.domain.Employee;
import com.test.provider.DepartmentProvider;
public interface DepartmentMapper {
    @SelectProvider(type = DepartmentProvider.class, method = "selectDepartmentByIdSql")
    public Department selectDepartmentById(@Param("depid") int depid);
}
```

这里我们使用@SelectProvider 来指定一个类及其内部的方法，用来提供需要执行的 SQL 语句。动态 sqlProvider 方法可以接收以下参数中的一种：

- 无参数
- 和映射器 mapper 接口的方法同类型的参数
- java.util.Map

如果映射器 mapper 接口有多个输入参数，我们可以使用参数类型为 java.util.Map 的方法作为 SQLprovider 方法。然后映射器 mapper 接口方法所有的输入参数将会被放到 Map 中，以 param1、param2 等作为 key，将输入参数按序作为 value。也可以使用你喜欢的名称作为 key 值，你可以使用@Param("depid")方式的注解标在 mapper 接口参数的名称前进行更改。

上面使用字符串拼接的方法构建 SQL 语句是非常困难的，并且容易出错。所以 MyBaits 提供了一个 SQL 工具类不使用字符串拼接的方式，简化构造动态 SQL 语句。现在，让我们看看如何使用 org.apache.ibatis.jdbc.SQL 工具类来准备相同的 SQL 语句，代码如下所示：

```
package com.test.provider;
import java.util.Map;
```

```
import org.apache.ibatis.jdbc.SQL;
public class DepartmentProvider {
    public String selectDepartmentByIdSql(Map<String, Object> parameters) {
        final int depid = Integer.valueOf(parameters.get("depid").toString());
        return new SQL() {
            {
                SELECT("dep.depid, dep.depname ");
                FROM("department dep ");
                WHERE("dep.depid =" + depid);
            }
        }.toString();
    }
}
```

SQL 工具类也提供了其他的方法来表示 JOIN、ORDER_BY、GROUP_BY 等。

让我们看一个使用 LEFT_OUTER_JOIN 的例子的代码：

```
package com.test.provider;
import java.util.Map;
import org.apache.ibatis.jdbc.SQL;
public class DepartmentProvider {
    public String selectDepartmentByIdSql(Map<String, Object> parameters) {
        return new SQL() {
            {
                SELECT("dep.depid, dep.depname");
                SELECT("emp.empid, emp.empname");
                FROM("department dep");
                LEFT_OUTER_JOIN("employee emp ON dep.depid = emp.depid");
                WHERE("dep.depid =  #{depid}");
            }
        }.toString();
    }
}
```

mapper 接口类方法代码：

```
@SelectProvider(type = DepartmentProvider.class, method = "selectDepartmentByIdSql2")
@ResultMap("com.test.mapper.DepartmentMapper.departmentResultMap")
public Department selectDepartmentById(@Param("depid")  int depid);
```

上面的 resultMap 可以配置在 XML 映射文件里，代码如下：

```
<resultMap id = "employeeResultMap" type = "com.test.domain.Employee">
        <id property = "empid" column = "empid" />
```

```
        <result property = "empname" column = "empname" />
    </resultMap>
    <resultMap id = "departmentResultMap" type = "com.test.domain.Department">
        <id property = "depid" column = "depid" />
        <result property = "depname" column = "depname" />
        <collection property = "employees" resultMap = "employeeResultMap"></collection>
    </resultMap>
```

3.5.9　@InsertProvider 动态插入语句

使用@InsertProvider 注解创建动态的 insert 语句，代码如下所示：

```
import java.util.Map;
import org.apache.ibatis.jdbc.SQL;
import com.test.domain.Department;
public class DepartmentProvider {
    public String insertDepartmentSql(final Department department) {
        return new SQL() {
            {
                INSERT_INTO("department");
                if (department.getDepname() != null)
                {
                    VALUES("depname", "#{depname}");
                }
            }
        }.toString();
    }
}
```

mapper 接口类方法代码如下：

```
    @InsertProvider (type = DepartmentProvider.class, method = "insertDepartmentSql")
    @Options(useGeneratedKeys = true, keyProperty = "depid")
    public int insertDepartment(Department department);
```

3.5.10　@UpdateProvider 动态更新语句

可以通过@UpdateProvider 注解创建 update 语句，代码如下所示：

```
package com.test.provider;
import java.util.Map;
import org.apache.ibatis.jdbc.SQL;
import com.test.domain.Department;
```

```
public class DepartmentProvider {
    public String updateDepartmentSql(final Department department) {
        return new SQL() {
            {
                UPDATE("department");
                if (department.getDepname() != null)
                {
                    SET("depname = #{depname}");
                }
                WHERE("depid = #{depid}");
            }
        }.toString();
    }
}
```

mapper 接口类方法代码如下所示：

```
@UpdateProvider(type = DepartmentProvider.class, method = "updateDepartmentSql")
    public int updateDepartment(Department department);
```

3.5.11 @DeleteProvider 动态删除语句

可以使用@DeleteProvider 注解创建动态的 delete 语句，代码如下所示：

```
package com.test.provider;
import java.util.Map;
import org.apache.ibatis.jdbc.SQL;
import com.test.domain.Department;
public class DepartmentProvider {
    public String deleteDepartmentSql(Map<String, Object> parameters) {
        return new SQL() {
            {
                DELETE_FROM("department");
                WHERE("depid = #{depid}");
            }
        }.toString();
    }
}
```

mapper 接口类方法：

```
    @DeleteProvider(type = DepartmentProvider.class, method = "deleteDepartmentSql")
    public int deleteDepartment(@Param("depid")int depid);
```

3.6　使用 MyBatis Generator 自动创建代码

MyBatis Generator (MBG) 是一个 MyBatis 的代码生成器。它可以生成 MyBatis 各个版本的代码，它根据数据库的表(或多个表)生成可以用来访问(多个)表的基础对象。这样和数据库表进行交互时不需要创建对象和配置文件。MBG 解决了对数据库操作有很大影响的一些简单的 CRUD(插入、查询、更新、删除)操作，但仍然需要对联合查询和存储过程手写 SQL 和对象。关于 MyBatis-Generator 的下载可以到这个地址：https://github.com/ mybatis/generator/releases，由于这里使用的是 mysql 数据库，因此需要再准备一个连接 MySQL 数据库的驱动 jar 包、MyBatis 框架的 jar 包以及 MyBatis 生成器 jar 包。包准备好之后有个 generatorConfig.xml 是需要我们来配置的，配置代码如下所示：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE generatorConfiguration
  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
  "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>
<!-- 数据库驱动-->
    <classPathEntry  location = "mysql-connector-java-5.1.40-bin.jar"/>
    <context id = "DB2Tables"  targetRuntime = "MyBatis3">
        <commentGenerator>
            <property name = "suppressDate" value = "true"/>
            <!-- 是否去除自动生成的注释 true：是，false：否 -->
            <property name = "suppressAllComments" value = "true"/>
        </commentGenerator>
        <!--数据库链接URL，用户名、密码 -->
        <jdbcConnection driverClass = "com.mysql.jdbc.Driver" connectionURL = "jdbc:mysql:///mydb" userId =
                                    "root" password = "root">
        </jdbcConnection>
        <javaTypeResolver>
            <property name = "forceBigDecimals" value = "false"/>
        </javaTypeResolver>
        <!-- 生成模型的包名和位置-->
        <javaModelGenerator targetPackage = "com.test.domain" targetProject = "src">
            <property name = "enableSubPackages" value = "true"/>
            <property name = "trimStrings" value = "true"/>
        </javaModelGenerator>
        <!-- 生成映射文件的包名和位置-->
        <sqlMapGenerator targetPackage = "com.test.mapper" targetProject = "src">
            <property name = "enableSubPackages" value = "true"/>
```

```
        </sqlMapGenerator>
        <!-- 生成DAO的包名和位置-->
        <javaClientGenerator type = "XMLMAPPER" targetPackage = "com.test.dao" targetProject = "src">
            <property name = "enableSubPackages" value = "true"/>
        </javaClientGenerator>
        <!-- 要生成的表 tableName是数据库中的表名或视图名，domainObjectName是实体类名-->
        <table tableName = "user" domainObjectName = "User" enableCountByExample = "false"
enableUpdateByExample = "false" enableDeleteByExample = "false" enableSelectByExample = "false"
selectByExampleQueryId = "false"></table>
    </context>
</generatorConfiguration>
```

当以上这些完成之后，只需要打开控制台，进入 lib 目录下，建立 src 文件夹，再执行脚本：Java -jar mybatis-generator-core-1.3.5.jar -configfile generatorConfig.xml –overwrite，如果成功生成配制文件，将在 src 下找到相应的实体类、接口、配置文件。

本章小结

本章是 MyBatis 的重点章节，在这章中我们学习了怎样使用映射器配置文件书写 SQL 映射语句，讨论了如何配置简单的语句，一对一以及一对多关系的语句，以及怎样使用 resultMap 进行结果集映射。我们还了解了如何构建动态 SQL 语句，学习了怎样使用注解书写 SQL 映射语句。探讨了怎样使用注解来构建动态 SQL 语句，最后我们学习了使用 MyBatis Generator 自动创建实体类、接口、配置文件。

练习题

一、选择题

1. 在为 SQL 文件配置映射接口时，以下说法错误的是(　　)。

A．Java 文件的接口名称可自由定义

B．SQL 的语句 id 必须和接口中的方法名相对应

C．SQL 的语句 parameterType 必须和接口中的参数类型相对应

D．SQL 的语句 resultType 必须和接口中的返回值相对应

2．下面 4 种 Mybatis 标签中，更新操作是(　　)。

A．<select id = "findAllUser" resultType = "hashmap">

B．<insert id = "insertUser" parameterType = "domain.User">

C．<update id = "updateUser" parameterType = "domain.User">

D．<delete id = "delUser" parameterType = "String">

3. 针对如下两种<mapper>配置导入 SQL 映射语句，正确的选择是(　　)。

(1) <mapper resource = "Mapper/UserMapper.xml"/>。

(2) <mapper file = "file:///var/Mapper/UserMapper.xml"/>。

A．都正确　　B．只有(1)正确　　C．只有(2)正确　　D．都不正确

4. 在为 SQL 文件配置映射接口时，以下说法正确的是(　　)。

A．Java 文件的接口名称可自由定义

B．SQL 的语句 id 必须和接口中的方法名相对应

C．SQL 的语句 parameterType 的类型可自由定义

D．SQL 的语句 resultType 和接口中的返回值可不一致

5．下面 4 种 Mybatis 标签中，删除操作是(　　)。

A．<select id = "findAllUser" resultType = "hashmap">

B．<insert id = "insertUser" parameterType = "domain.User">

C．<update id = "updateUser" parameterType = "domain.User">

D．<delete id = "delUser" parameterType = "String">

二、填空题

1．一个 select SQL 语句可以在_____元素的映射器 XML 配置文件中配置。

2．一个 insert SQL 语句可以在_____元素的映射器 XML 配置文件中配置。

3．一个 update SQL 语句可以在_____元素的映射器 XML 配置文件中配置。

4．一个 delete SQL 语句可以在_____元素的映射器 XML 配置文件中配置。

5．<select>元素里的_____属性表示这个映射在这个命名空间下唯一的标识符，可被其他语句引用。

6．<select>元素里的_____属性表示这个映射传入参数的类型，传给此语句的参数的完整类名或别名。

7．<select>元素里的_____属性表示这个映射返回值类型的完整类名或别名。

8．<select>元素里的_____属性表示引用的外部定义的结果集映射，它与_____不能同时使用。

9．<resultMap>里_____表示 JavaBean 里映射数据库列的字段或属性，_____数据库的列名或者列标签别名，与传递给 resultSet.getString(columnName)的参数名称相同。

10．resultMap 除了常用标签外还提供了实例化注入_____元素。

11．我们可以从另外一个<resultMap>拓展出一个新的<resultMap>，这样，原先的属性映射可以_____过来。

12．<resultMap>使用_____元素处理“has-one”(一对一)这种类型关系。

13. <resultMap>可以使用_____元素将一对多类型的结果映射到一个对象集合上。

14. 和一对一映射一样，我们可以使用嵌套结果_____和嵌套_____语句两种方式映射实现一对多映射。

15. 要开启二级缓存，只需要在 SQL 映射文件中加入简单的一行_____。

16. 在某一个命名空间里可以使用<cache>元素配置或者刷新缓存。但有可能您想要在不同的命名空间里共享同一个缓存配置或者实例。在这种情况下，可以使用_____元素来引用另外一个缓存。

17. MyBatis3 使用基于强大的_____表达式消除了大部分元素。

18. 动态 SQL 里_____就是简单的条件判断，利用它我们可以实现某些简单的条件选择。

19. 有时候我们不想应用所有的条件，而是想从多个选项中选择一个。与 Java 中的 switch 语句相似，MyBatis 提供了一个_____元素，_____元素表示当其中的条件满足的时候就输出其中的内容，当所有的条件都不满足的时候就输出_____中的内容。

20. _____元素的主要功能是可以在自己包含的内容前加上某些前缀，也可以在其后加上某些后缀，与之对应的属性是_____和_____；也可以把尾部的某些内容覆盖，对应的属性是_____ 和_____；正因为其有这样的功能，所以我们也可以非常简单地利用它来代替_____元素的功能。

21. _____元素主要是用在更新操作的时候，它的主要功能和 where 元素其实是差不多的。

22. _____主要用在构建 in 条件中，它可以在 SQL 语句中迭代一个集合。

23. 可以在 mapper 接口方法上使用_____注解来定义一个 SELECT 映射语句。

24. 对于支持 AUTO_INCREMENT 的数据库，可以使用_____注解的 userGeneratedKeys 和 keyProperty 属性让数据库产生 auto_increment(自增长)列的值。

25. 可以在 mapper 接口方法上使用_____注解来定义一个 update 映射语句。

26. 可以在 mapper 接口方法上使用_____注解来定义一个 delete 映射语句。

27. 可以将查询结果通过别名或者是_____注解与 JavaBean 属性映射起来。

28. MyBatis 提供了_____注解来使用嵌套 select 语句(Nested-Select)加载一对一关联查询数据。

29. MyBatis 提供了_____注解，用来使用嵌套 select 语句加载一对多关联查询。

30. MyBatis 使用_____注解创建动态的 select 语句。

31. MyBatis 使用_____注解创建动态的 insert 语句。

32. MyBatis 使用_____注解创建

动态的 update 语句。

33．MyBatis 使用_____注解创建动态的 delete 语句。

三、问答题

1．MyBatis 的接口绑定有什么好处？

2．#{...}和${...} 的区别是什么？

3．MyBatis 如何加载一个联合查询？

4．简述“N+1”问题的产生。

5．以学生(stuNo，stuName，classId)和班级(classId，className)为例，说明 MyBatis 中一对多的查询的配置过程。

6. 简述 MyBatis 编程和 JDBC 编程的不同之处。

第四章　Spring 核心技术

Spring 是一个功能强大的开源框架，它为企业级开发提供了丰富的功能，但是这些功能的底层都依赖于它的两个核心特性，也就是依赖注入(Dependency Injection，DI)和面向切面编程(Aspect-Oriented Programming，AOP)。

本章快速介绍了 Spring 框架，包括 Spring DI 和 AOP 的概况，以及它们是如何帮助读者解耦应用组件的；在“装配 Bean”中，我们将深入探讨如何将应用中的各个组件拼装在一起，读者将会看到 Spring 所提供的自动配置、基于 Java 的配置、XML 配置以及高级装配；在“面向切面的 Spring”中，展示如何使用 Spring 的 AOP 特性把系统级的服务(例如事务、日志、审计)从它们所服务的对象中解耦出来。

本章知识要点

- Spring 开发环境的搭建；
- 依赖注入；
- 面向切面编程。

4.1　Spring 简介

Spring 是一个开源框架，最早由 Rod Johnson 创建，并在《Expert One-on-One：J2EE Design and Development》(http://amzn.com/076454385)这本著作中进行了介绍。Spring 是为了解决企业级应用开发的复杂性而创建的，使用 Spring 可以让简单的 POJO 实现之前只有 EJB 才能完成的事情。但 Spring 不仅仅局限于服务器端开发，任何 Java 应用都能在简单性、可测试性和松耦合等方面从 Spring 中获益。

4.1.1　Sping 的核心模块

Spring 框架大约由 20 个功能模块组成，这些模块分别被分组到 Core Container、Data Access/Integration、Web、AOP(面向切面的编程)、Instrumentation、Messaging 和 Test 中，其结构如图 4-1 所示。

组成 Spring 框架的每个模块(或组件)都可以单独存在，或者与其他一个或多个模块联合实现。每个模块的功能如下：

(1) Spring Core：核心容器提供了 Spring 的基本功能。核心容器的核心功能是用 IoC 容器来管理类的依赖关系的。Spring 采用的模式是调用者不理会被调用者的实例的创建，由 Spring 容器负责被调用者实例的创建和维护，需要时注入给调用者。这是目前最优秀的解耦模式。

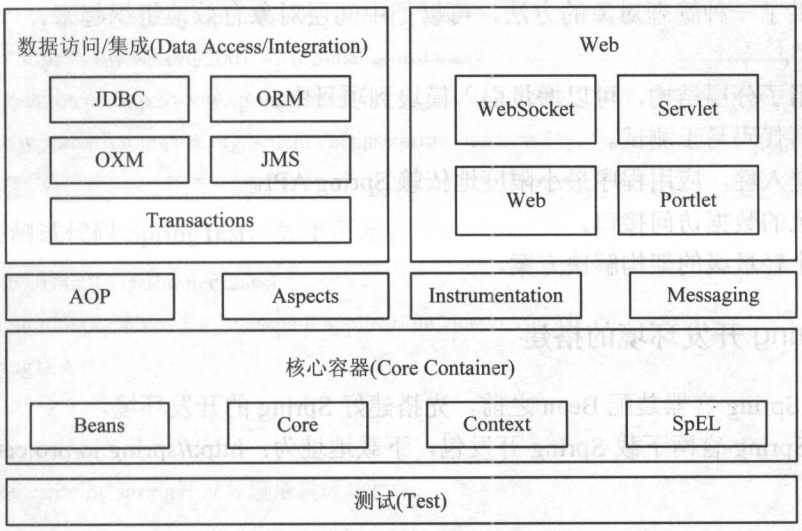

图 4-1　Spring 框架概览

(2) Spring AOP：Spring 的 AOP 模块提供了面向切面编程的支持。Spring AOP 采用的是纯 Java 实现方案。Spring AOP 采用基于代理的 AOP 实现方案，AOP 代理由 IoC 容器负责生成、管理。依赖关系也一并由 IoC 容器管理。尽管如此，Spring IoC 容器并不依赖于 AOP，这样我们可以自由选择是否使用 AOP。

(3) Spring ORM：提供了与多个第三方持久层框架的良好整合。

(4) Spring DAO：Spring 进一步简化 DAO 开发步骤，能以一致的方式使用数据库访问技术，用统一的方式调用事务管理，避免具体的实现侵入业务逻辑层的代码中。

(5) Spring Context：这是一个配置文件，为 Spring 提供上下文信息，并提供了框架式的对象访问方法。Context 为 Spring 提供了一些服务支持，如国际化(i18n)、电子邮件、校验和调度功能。

(6) Spring Web：提供了基础的针对 Web 开发的集成特性，例如多方文件上传，利用 Servlet listeners 进行 IoC 容器初始化和针对 Web 的 applicationContext。

(7) SpringMVC：提供了 Web 应用的 MVC 实现。Spring 的 MVC 框架并不是仅仅提供一种传统的实现方案，它提供了一种清晰的分离模型，在领域模型代码和 Web form 之间。并且，还可以借助 Spring 框架的其他特性。

(8) 消息传递：Spring 框架包含的 spring-messaging 模块带有一些来自诸如 Message、MessageChannel、MessageHandler 等 Spring Integration 对象的关键抽象，它们被用于基于消息传递应用的服务基础。这个模块映射包含了一组用于消息映射的方法注释，类似于基于编程模式的 SpringMVC 注解。

4.1.2　Spring 框架的优势

Spring 是一个开源框架，是为了解决企业应用程序开发的复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许您根据项目需要选择使用 Spring 模块组件，同时为 J2EE 应用程序开发提供集成的框架。关于 Spring 框架可概括如下：

(1) 提供了一种管理对象的方法，可以把中间层对象有效地组织起来。一个完美的框架“黏合剂”。

(2) 采用了分层结构，可以增量引入模块到项目中。

(3) 编写代码易于测试。

(4) 非侵入性，应用程序最小限度地依赖 Spring API。

(5) 一致的数据访问接口。

(6) 一个轻量级的架构解决方案。

4.1.3 Spring 开发环境的搭建

在讲述 Spring 容器装配 Bean 之前，先搭建好 Spring 的开发环境。

(1) 在 Spring 官网下载 Spring 开发包，下载地址为：http://spring.io/projects，如图 4-2 所示。

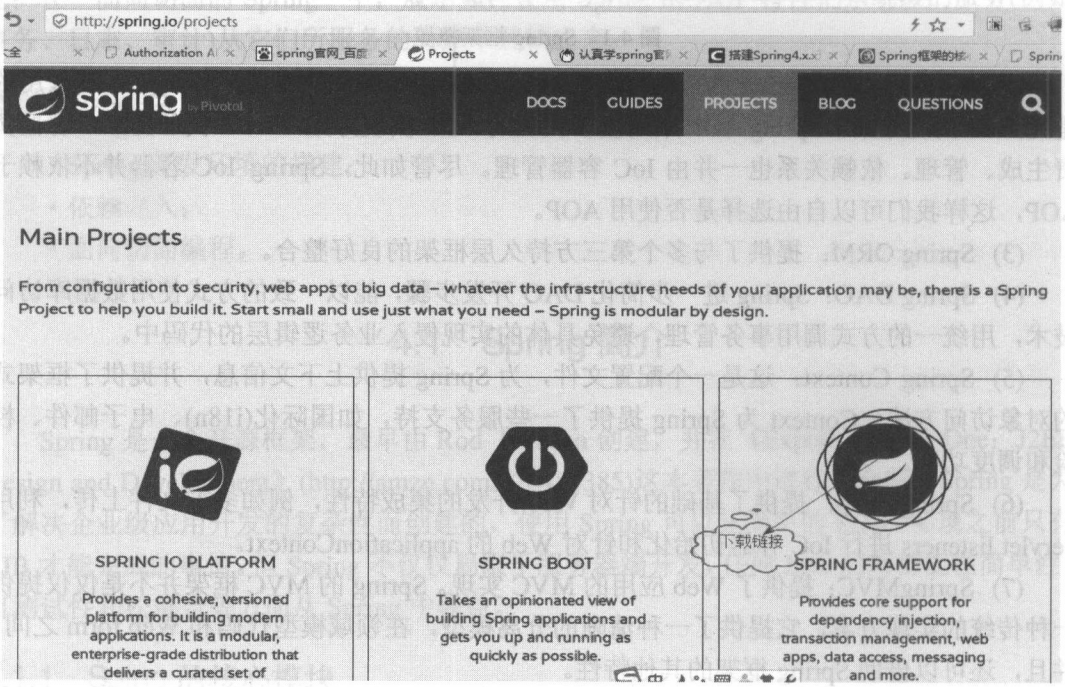

图 4-2 Spring 开发包下载页面

(2) 在 Eclipse 中新建 Java Web 项目 SpringTest，解压 Spring 开发包，在子文件夹 libs 中复制如下 jar 包：

spring-core-4.3.0.RELEASE.jar

spring-beans-4.3.0.RELEASE.jar

spring-context-4.3.0.RELEASE.jar

spring-expression-4.3.0.RELEASE.jar

(3) 在项目 src 文件夹下创建 Spring 配置文件 applicationContext.xml，代码如下所示：

```
<?xml version = "1.0" encoding = "UTF-8"?>
```

```
<beans xmlns = "http://www.springframework.org/schema/beans"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
</beans>
```

(4) 编写测试代码 SpringTest，如下所示：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:applicationContext.xml" })
public class SpringTest {
    @Test
    public void test() {
        System.out.println("spring开发环境搭建成功！ ");
    }
}
```

运行 test 方法结果如图 4-3 所示。

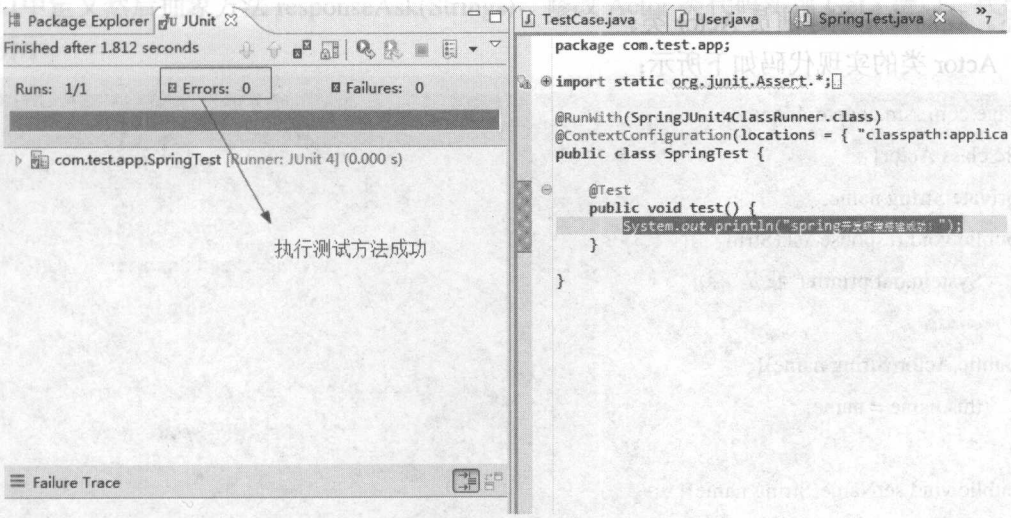

图 4-3　运行测试 test 方法的运行结果

从运行测试结果看无错误，表示 Spring 开发环境搭建成功。

4.2　控制反转(IoC)

4.2.1　IoC 的基本概念

控制反转(Inversion of Control，IoC)是一个重要的面向对象编程的法则，用来削减计算机程序的耦合问题，也是轻量级的 Spring 框架的核心。DI(依赖注入)其实就是 IoC 的另

外一种说法，DI 是由 Martin Fowler 在 2004 年初的一篇论文中首次提出的。他总结：控制的什么被反转了？就是获得依赖对象的方式被反转了。

组件之间的依赖关系由容器在运行期决定，形象地说，依赖注入即由容器动态地将某个依赖关系注入组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要进行简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。理解依赖注入的关键是“谁依赖谁，为什么需要依赖，谁注入谁，注入了什么”，下面我们来深入分析这些概念。

(1) 谁依赖于谁：当然是应用程序依赖于 IoC 容器；

(2) 为什么需要依赖：应用程序需要 IoC 容器来提供对象需要的外部资源；

(3) 谁注入谁：很明显是 IoC 容器注入应用程序某个对象，该对象是应用程序依赖的对象；

(4) 注入了什么：就是注入某个对象所需要的外部资源(包括对象、资源、常量数据)。

现在我们通过一个实例来理解 DI，电影《墨攻》的一个场景：当刘德华所饰演的墨者革离角色到达梁国都城下，城上梁国守军问道：“来者何人？”刘德华回答：“墨者革离！”革离是电影《墨攻》的男主角，我们不妨通过一个 Java 类为这个“城门叩问”的场景进行描述。我们首先创建演员 Actor 类。

Actor 类的实现代码如下所示：

```
package com.ssm.chapter4.ioc;
public class Actor{
    private String name;
    public void responseAsk(String s){
        System.out.println("我是"+s);
    }
    public Actor(String name){
        this.name = name;
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
}
```

Actor 类定义一个演员姓名的成员变量 name，添加了 name 属性的 getter 和 setter 方法，定义一个公有的构造方法，初始化演员的名字。

我们还要定义一个剧本类 MoAttack(墨攻)，代码如下所示：

```
package com.ssm.chapter4.ioc;
```

```
public class MoAttack {
    public void cityGateAsk(){
        Actor actor = new Actor("刘德华");
        actor.responseAsk("墨者革离");
    }
}
```

我们会发现在 MoAttack 类中的 cityGateAsk()方法中，作为具体角色墨者革离的饰演者刘德华对象直接侵入到剧本中，使剧本和演员直接耦合在一起，如图 4-4 所示。

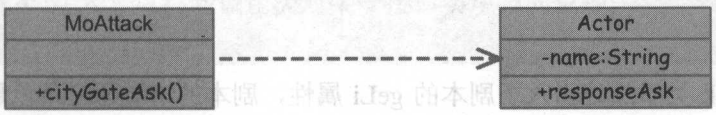

图 4-4　剧本和演员直接耦合

一个明智的编剧在剧情创作时应围绕故事中的角色进行，而不应考虑角色的具体饰演者，这样才可能在剧本投拍时自由地遴选任何适合的演员，而非绑定在演员刘德华一人身上。通过以上的分析，我们知道需要为该剧本主人公“革离”定义一个接口 GeLi，在 GeLi 接口中定义然后抽象方法 responseAsk(Strings)，修改 Actor 类代码实现 GeLi 接口，代码如下所示：

```
package com.ssm.chapter4.ioc;
public interface GeLi {
    void responseAsk(String s);
}
public class Actor implements GeLi {
    private String name;
    @override
    public void responseAsk(String s){
        System.out.println("我是"+s);
    }
    public Actor(String name){
        this.name = name;
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
}
//MoAttack：引入剧本角色
```

```
package com.ssm.chapter4.ioc;
public class MoAttack {
    private GeLi geLi;
    public setGeLi(Actor actor){
        this. geLi = actor;
    }
    public void cityGateAsk(){
        geLi.responseAsk("墨者革离");
    }
}
```

在 MoAttack 类添加处引入了剧本的 geLi 属性，剧本的情节通过角色展开，在拍摄时角色通过 setGeLi()方法注入 Actor 对象扮演 GeLi 角色就可以了。因此墨攻、革离、演员三者的类图关系如图 4-5 所示。

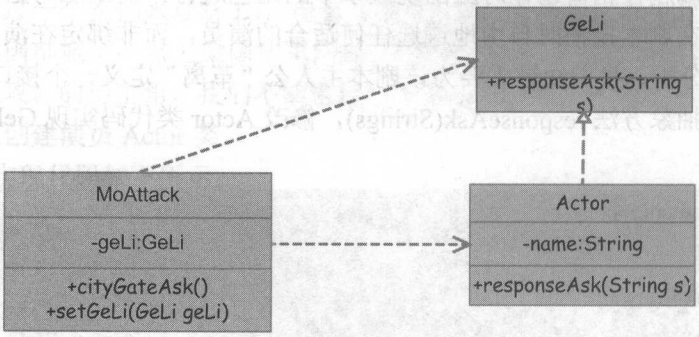

图 4-5　引入角色接口的关系

从图 4-5 中，我们可以看出 MoAttack 同时依赖于 GeLi 接口和 Actor 类，并没有达到我们所期望的剧本仅依赖于角色的目的。但是角色最终必须通过具体的演员才能完成拍摄，如何让演员和剧本无关而又能完成 GeLi 的具体动作呢？当然是在影片投拍时，导演将 Actor 对象(刘德华)安排在 GeLi 的角色上，也可以安排其他演员扮演 GeLi 角色，于是导演将剧本、角色、饰演者装配起来，如图 4-6 所示。

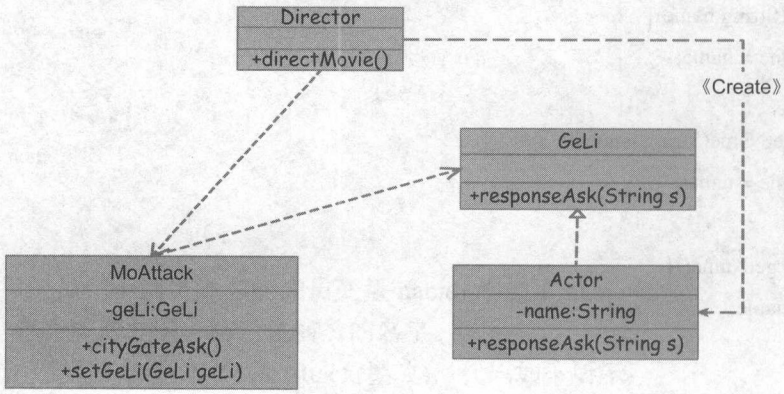

图 4-6　剧本和演员解耦

通过引入导演，使剧本和具体饰演者解耦，对应到软件中，导演像是一个装配器，安排演员表演具体的角色。现在我们来讲解 IoC 的概念，IoC(Inverse of Control)的字面意思是控制反转，它包括两个内容：一是控制，二是反转。那到底是什么东西的“控制”被“反转”了呢？对应到前面的例子，“控制”是指选择 GeLi 角色扮演者的控制权；“反转”是指这种控制权从《墨攻》剧本中移除，转交到导演的手中。对于软件来说，即是某一接口具体实现类的选择控制权从调用类中移除，转交给第三方决定，因为 IoC 概念隐晦，所以业界曾进行了广泛的讨论，最终软件界的泰斗级人物 Martin Fowler 提出了 DI(依赖注入：Dependency Injection)的概念用以代替 IoC，即让调用类对某一接口实现类的依赖关系由第三方(容器或协作类)注入，以移除调用类对某一接口实现类的依赖。

4.2.2　依赖注入的类型

从注入方法上看，可以将依赖注入划分为三种类型：构造方法注入、属性注入和接口注入。Spring 支持构造方法注入和属性注入。下面我们继续说明这三种注入方法的区别。

1. 构造方法注入

通过构造函数注入革离扮演者，代码如下所示：

```
package com.ssm.chapter4.ioc;
public class MoAttack
{
    GeLi geLi;
    public MoAttack(GeLi geLi){
        this.geLi = geLi;
    }
    public void cityGateAsk(){
        geLi.responseAsk("墨者革离");
    }
}
```

MoAttack 的构造函数不关心具体是谁扮演革离这个角色，只要革离角色的扮演者按剧本要求完成相应的表演，角色的扮演者由导演选定，通过构造方法注入扮演者即可，代码如下所示：

```
package com.ssm.chapter4.ioc;
public class Director {
    public void Direct(){
        //①指定GeLi角色的扮演者
        GeLi geLi = new Actor("刘德华");
        //②注入具体扮演者到剧本中
        MoAttack moAttack = new MoAttack(geLi);
        moAttack.cityGateAsk();
```

```
    }
}
```

导演安排刘德华饰演革离的角色，将刘德华“注入”到墨攻的剧本 MoAttack 中，然后开始“城门叩问”剧情的演出工作。

2. 属性注入

我们也可以通过属性注入 GeLi 扮演者，使用 Setter 方法注入革离扮演者，代码如下所示：

```
public class MoAttack {
    private GeLi geLi;
    public void cityGateAsk(){
        geLi.responseAsk("墨者革离");
    }
    //属性方法注入GeLi扮演者
    public void setGeLi(GeLi geLi) {
        this.geLi = geLi;
    }
}
```

从上面代码可知，当程序需要 GeLi 的扮演者的时候，通过 MoAttack 的 setGeLi 的方法注入 GeLi 扮演者。相比构造方法注入对象更加灵活。

3. 接口注入

将调用类所有依赖注入的方法抽取到一个接口中，调用类通过实现该接口提供相应的注入方法。为了采取接口注入的方式，必须先声明一个 ActorArrangable 接口，代码如下所示：

```
public interface ActorArrangable {
    void injectGeLi(GeLi geLi);
}
```

MoAttack 实现 ActorArrangable 接口，注入 GeLi 扮演者。MoAttack 通过接口注入 GeLi 扮演者，代码如下所示：

```
public class MoAttack implements ActorArrangable {
    private GeLi geLi;
    public void cityGateAsk(){
        geLi.responseAsk("墨者革离");
    }
    //属性方法注入GeLi扮演者
    public void setGeLi(GeLi geLi) {
        this.geLi = geLi;
    }
```

```
    //实现接口方法，注入扮演GeLi角色的演员
    @Override
    public void injectGeLi(GeLi geLi) {
        this.geLi = geLi;

    }
}
```

Director 通过 ActorArrangable 的 injectGeli()方法完成扮演者的注入工作。通过接口方法注入革离扮演者，代码如下所示：

```
package com.ssm.chapter4.ioc;
public class Director
{
  public void direct(){
    GeLi geli = new Actor("刘德华");
    MoAttack moAttack = new MoAttack();
    //通过接口方法注入GeLi扮演者
    moAttack.injectGeLi (geli);
    moAttack.cityGateAsk();
  }
}
```

由于通过接口注入需要额外声明一个接口，增加了类的数目，而且它的效果和属性注入并无本质区别，因此我们不提倡采用这种方式。

MoAttack 和 Actor 实现了解耦，MoAttack 无需关注角色实现类的实例化工作，第三方中介机构在程序领域即是一个第三方的容器，它帮助完成类的初始化与装配工作，让开发者从这些底层实现类的实例化、依赖关系装配等工作中脱离出来，专注于更有意义的业务逻辑开发工作。这无疑是一件令人向往的事情，Spring 就是这样的一个容器，它通过配置文件或注解描述类和类之间的依赖关系，自动完成类的初始化和依赖注入的工作。下面是 Spring 配置文件对以上实例进行配置的片段。

applicationContext.xml 文件片段：

```
    <bean name = "geLi" class = "com.ssm.chapter4.ioc.Actor">
    <property name = "name" value = "liudehua"/>
    </bean>
    <bean name = "MoAttack" class = "com.ssm.chapter4.ioc.MoAttack">
        <property name = "geLi" ref = "geLi"></property>
</bean>
```

通过实例化容器对象，Spring 根据配置文件的描述信息，自动实例化 Bean 并完成依赖关系的装配，从容器中即可返回准备就绪的 Bean 实例，后续可直接使用之。

4.3 Bean 的装配

在基于 Sping 的应用中，容器可以创建、装配和配置应用组件，在容器中的对象称为 Bean。

4.3.1 Spring 装配 Bean 的方案

Spring 容器负责创建应用程序中的 Bean，并通过 DI 来协调这些对象之间的关系。但是，作为开发人员，你需要告诉 Spring 要创建哪些 Bean，并且如何将其装配在一起。当描述 Bean 如何进行装配时，Spring 具有非常大的灵活性，它提供了三种主要的装配机制：

(1) 在 XML 中进行显式配置。

(2) 在 Java 中进行显式配置。

(3) 隐式的 Bean 发现机制和自动装配。

用户可根据自己的喜好选择装配方式。

4.3.2 Spring IoC 容器

1. Spring 容器介绍

Spring 有两个核心接口：BeanFactory 和 ApplicationContext，其中 ApplicationContext 是 BeanFactory 的子接口。它们都可以代表 Spring 容器，Spring 容器是生成 Bean 实例的工厂，并且管理容器中的 Bean。

在应用中的所有组件，都处于 Spring 的管理下，都被 Spring 以 Bean 的方式管理，Spring 负责创建 Bean 实例，并管理它们的生命周期。Bean 在 Spring 容器中运行，无需考虑 Spring 容器的存在，一样可以接受 Spring 的依赖注入，包括 Bean 属性的注入、协作者的注入、依赖关系的注入等。

在 Spring 配置文件中配置好所有的组件，当 ApplicationContex 创建和初始化完成后，就可以执行系统或应用程序了，Spring 框架运行的高级视图如图 4-7 所示。

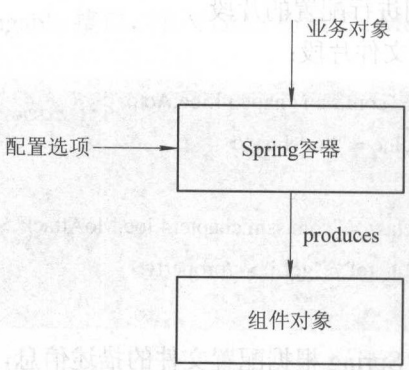

图 4.7 Spring IoC 容器

2. 基于 XML 配置的元数据

基于 XML 配置的代码如下所示：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop = "http://www.springframework.org/schema/aop"
    xmlns:context = "http://www.springframework.org/schema/context"
    xmlns:tx = "http://www.springframework.org/schema/tx"
    xsi:schemaLocation = "http://www.springframework.org/schema/beans
                          http://www.springframework.org/schema/beans/spring-beans.xsd
                          http://www.springframework.org/schema/aop
                          http://www.springframework.org/schema/aop/spring-aop.xsd
                          http://www.springframework.org/schema/context
                          http://www.springframework.org/schema/context/spring-context.xsd
                          http://www.springframework.org/schema/tx
                          http://www.springframework.org/schema/tx/spring-tx.xsd">
    <context:component-scan base-package = "com.ssm.chapter4"></context:component-scan>
    <bean name = "geLi" class = "com.ssm.chapter4.ioc.LiuDeHua"></bean>
    <bean name = "moAttack" class = "com.ssm.chapter4.ioc.MoAttack">
        <property name = "geLi" ref = "geLi"></property>
    </bean>
</beans>
```

以上代码就是 Spring 基于 XML 配置的元数据。

3. 实例化容器对象

实例化容器对象的代码如下所示：

```
ApplicationContext applicationContext = new ClassPathXmlApplicationContext(
                    new String[]{"applicationContext.xml"});
```

4. 使用容器对象获得 Bean 对象

我们可以从容器中获取一个 Bean。通过 applicationContext 的 getBean 方法，上面 Spring 配置文件中已经有一个名为“geLi”的 Bean，可以通过下面的代码得到 geLi 对象：

```
ApplicationContext applicationContext = new ClassPathXmlApplicationContext(
                new String[]{"applicationContext.xml"});
GeLi geLi = (GeLi) applicationContext.getBean("geLi");
```

4.3.3　基于注解的 Bean 装配

1. 创建 Student 类

在 Student 类中添加@Component，声明 Student 对象是 Spring 容器管理的对象，代码

如下所示：

```
@Component
public class Student {
    private String ID;
    private String name;
    private int age;
    private String sex;
    public String getID() {
        return ID;
    }
    public void setID(String id) {
        ID = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getSex() {
        return sex;
    }
    public void setSex(String sex) {
        this.sex = sex;
    }
}
```

需要注意的就是 Student 类上使用了@Component 注解。这个简单的注解表明该类会作为组件类，并告知 Spring 要为这个类创建 Bean。

2. 组件扫描开启

组件扫描默认是不启用的。我们还需要显式修改 Spring 配置文件，从而命令它去寻找带有@Component 注解的类，并为其创建 Bean。如果使用 XML 来启用组件扫描，则可以使用 Spring context 命名空间的<context:component-scan>元素，设置属性 base-package，指

定扫描的包名。代码如下所示：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
       xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context = "http://www.springframework.org/schema/context"
       xsi:schemaLocation = "http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd>
<context:component-scanbase-package = "com.ssm.chapter4">
</context:component-scan>
</beans>
```

我们也可以使用@ComponentScan 注解开启组件扫描。代码如下所示：

```
package com.ssm.chapter4.autobean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
@Configuration
@ComponentScan
public class MyConfig {
}
```

3. 编写测试类

为了测试组件扫描的功能，我们创建一个简单的 JUnit 测试，它会创建 Spring 上下文，并判断 Student 是不是真的创建出来了。AutobeanTest 就是用来测试组件扫描去发现 Student 对象的，代码如下所示：

```
package com.ssm.chapter4.test;
import javax.annotation.Resource;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import com.ssm.chapter4.autobean.MyConfig;
import com.ssm.chapter4.autobean.Student;
@RunWith(SpringJUnit4ClassRunner.class)
//@ContextConfiguration(locations = { "classpath:applicationContext.xml" })
@ContextConfiguration(classes = MyConfig.class)
public class AutobeanTest {
```

```
    @Resource
    Student student;
    @Test
    public void test() {
        Assert.assertNotNull(student);
    }
}
```

运行测试方法 test()验证 Student 对象不为空，AutobeanTest 类在定义成员变量 Student 时，在其类型的前面添加@Resource，表示在 Spring 容器中查找 Student 类型的对象并初始化 Student 对象，如图 4-8 所示。

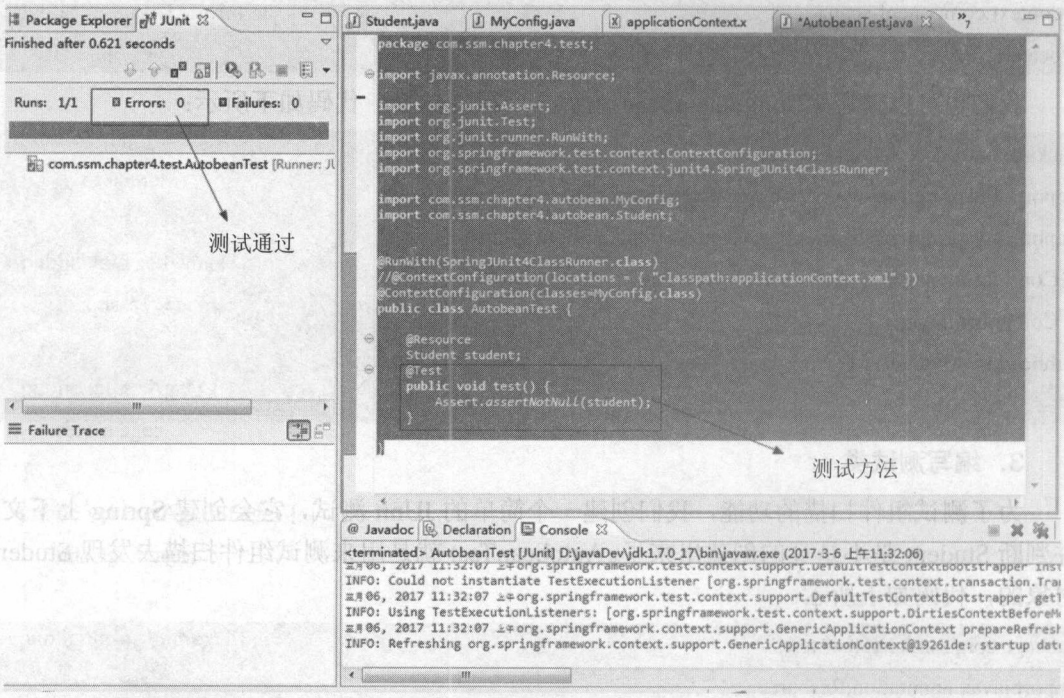

图 4-8　运行结果

4.4　面向切面编程(SOP)

4.4.1　面向切面编程简介

Spring AOP 是 Spring 框架的重要组成部分，面向切面编程(Aspect-Oriented Programming，AOP)是以另一个角度来考虑程序结构，通过分析程序结构的关注点来完善面向对象编程(OOP)。OOP 将应用程序分解成各个层次的对象，而 AOP 将程序分解成多个切面。Spring AOP 只实现了方法级别的连接点，在 J2EE 应用中，AOP 拦截到方法级别的

操作就已经足够了。在 Spring 中需要利用 spring AOP 实现为 IoC 和企业服务之间建立联系。AOP 可以说是对 OOP 的补充和完善。OOP 引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当需要为分散的对象引入公共行为的时候，OOP 则显得无能为力。也就是说，OOP 允许定义从上到下的关系，但并不适合定义从左到右的关系，例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。在 OOP 设计中，这种方案导致了大量代码的重复，不利于各个模块的重用。而 AOP 则将程序中的交叉业务逻辑(比如安全、日志、事务等)封装成一个切面，然后注入到目标对象(具体业务逻辑)中去。

AOP 技术的实现，主要分为两大类：一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态植入的方式，引入特定的语法创建“切面”，从而使编译器可以在编译期间织入有关“切面”的代码。

Spring 对面向切面编程提供了强有力的支持，通过它我们可以将业务逻辑从应用服务(如事务管理)中分离出来，实现了高内聚开发，应用对象只关注业务逻辑，不再负责其他系统问题(如日志、事务等)。Spring 支持用户自定义切面。

下面介绍 AOP 的一些术语。

切面(Aspect)：一个关注点的模块化，这个关注点可能会横切多个对象。事务管理是 J2EE 应用中一个关于横切关注点的很好的例子。在 Spring AOP 中，切面可以使用基于模式或者基于@Aspect 注解的方式来实现。

连接点(Joinpoint)：在程序执行过程中某个特定的点，比如某方法调用的时候或者处理异常的时候。在 Spring AOP 中，一个连接点总是表示一个方法的执行。

通知(Advice)：在切面的某个特定的连接点上执行的动作，其中包括了“around”、“before”和“after”等不同类型的通知(通知的类型将在后面部分进行讨论)。许多 AOP 框架(包括 Spring)都是以拦截器作通知模型，并维护一个以连接点为中心的拦截器链。

切入点(Pointcut)：匹配连接点的断言，通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行(例如，当执行某个特定名称的方法时)。切入点表达式如何和连接点匹配是 AOP 的核心，Spring 缺省使用 AspectJ 切入点语法。

引入(Introduction)：用来给一个类型声明额外的方法或属性(也被称为连接类型声明(inter-type declaration))。Spring 允许引入新的接口(以及一个对应的实现)到任何被代理的对象，例如，你可以使用引入来使一个 Bean 实现 IsModified 接口，以便简化缓存机制。

目标对象(Target Object)：被一个或者多个切面所通知的对象，也被称做被通知(advised)对象。既然 Spring AOP 是通过运行时代理实现的，这个对象永远是一个被代理(proxied)对象。

AOP 代理(AOP Proxy)：AOP 框架创建的对象，用来实现切面契约(例如通知方法执行等)。在 Spring 中，AOP 代理可以是 JDK 动态代理或者 CGLIB 代理。

织入(Weaving)：把切面连接到其他的应用程序类型或者对象上，并创建一个被通知的对象，这些可以在编译时(例如使用 AspectJ 编译器)、类加载时和运行时完成。Spring 和其他纯 Java AOP 框架一样，在运行时完成织入切面。

通知类型：分为前置通知、后置通知、异常通知、最终通知和环绕通知，具体定义如下：

前置通知(Before advice)：在某连接点之前执行的通知，但这个通知不能阻止连接点之前的执行流程(除非它抛出一个异常)。

后置通知(After returning advice)：在某连接点正常完成后执行的通知，例如，一个方法没有抛出任何异常，正常返回。

异常通知(After throwing advice)：在方法抛出异常退出时执行的通知。

最终通知(After (finally) advice)：当某连接点退出的时候执行的通知(不论是正常返回还是异常退出)。

环绕通知(Around Advice)：包围一个连接点的通知，如方法调用，这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它自己的返回值或抛出异常来结束执行。

4.4.2 通过切点选择连接点

正如之前所提过的，切点用于准确定位应该在什么地方应用切面的通知。通知和切点是切面最基本的元素。因此，了解如何编写切点非常重要。

在 Spring AOP 中，要使用 AspectJ 的切点表达式语言来定义切点。如果你已经很熟悉 AspectJ，那么在 Spring 中定义切点就非常自然。但是如果你一点都不了解 AspectJ，本小节我们将快速介绍如何编写 AspectJ 风格的切点。

关于 Spring AOP 的 AspectJ 切点，最重要的一点就是 Spring 仅支持 AspectJ 切点指示器(Pointcut Designator)的一个子集。让我们回顾一下，Spring 是基于代理的，而某些切点表达式是与基于代理的 AOP 无关的。表 4-1 列出了 Spring AOP 所支持的 AspectJ 切点指示器。

Spring 借助 AspectJ 的切点表达式语言来定义 Spring 切面，如表 4-1 所示。

表 4-1 AspectJ 指示器

AspectJ 指示器	描　述
arg()	限制连接点匹配参数为指定类型的执行方法
@args()	限制连接点匹配参数由指定注解标注的执行方法
execution()	用于匹配连接点的执行方法
this()	限制连接点匹配 AOP 代理的 Bean 引用为指定类型的类
target	限制连接点匹配目标对象为指定类型的类
@target()	限制连接点匹配特定的执行对象，这些对象对应的类要具有指定类型的注解
within()	限制连接点匹配指定的类型
@within()	限制连接点匹配指定注解所标注的类型(当使用 Spring AOP 时，方法定义在由指定的注解所标注的类里)
@annotation	限定匹配带有指定注解的连接点

上表展示的 Spring 支持的指示器，只有 execution 指示器是执行实际匹配的，其他的指示器都是用来限制匹配的。这说明 execution 指示器是我们在编写切点定义时最主要使用的指示器。在此基础上，我们使用其他指示器来限制所匹配的切点。

1. 编写切点

为了阐述切点，我们需要一个主题来定义切面的切点，为此我们定义一个接口 Performance：

```
package com.ssm.chapter4.aop;
public interface Performance {
    public void perform();
}
```

定义一个实现 Performance 的实现类 Performer，代码如下：

```
package com.ssm.chapter4.aop;
public class Performer implements Performance {
    public void perform(){
        System.out.println("正在表演！ ");
    }
}
```

假设我们编写调用 Performance 接口的 perform()时触发通知切点表达式，图 4-9 展示一个切点表达式，这个表达式在调用 Performance 接口的 perform()时触发通知。

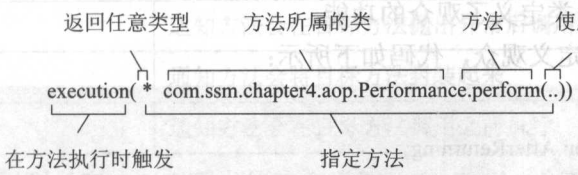

图 4-9　使用 AspectJ 切点表达式选择 Performance 的 perform()方法

2. 在切点中选择 Bean

除了表 4-1 所列的指示器外，Spring 还引入了一个新的 bean()指示器，它允许我们在切点表达式中使用 Bean 的 ID 来标识 Bean。bean()使用 Bean ID 或 Bean 名称作为参数来限制切点只匹配特定的 Bean。

首先在 application.xml 中注册一个 Bean 对象，代码如下所示：

```
<bean id = "waterBrother" class = "com.ssm.chapter4.ioc.Performer">
```

例如，考虑如下的切点：

```
execution( *  com.ssm.chapter4.aop.Performance.perform(..))
and bean('waterBrother')
```

在这里，我们希望在执行 Performance 的 perform()方法时应用通知，但限定 id 为“waterBrother”的 Bean。我们还可以运用非操作除了指定 ID 以外的 bean 应用通知。

```
execution( *  com.ssm.chapter4.aop.Performance.perform(..))
and ! bean('waterBrother')
```

我们已经讲解了编写切点相关的基础知识，下面继续了解编写通知和使用切点声明切面。

4.4.3 使用注解创建切面

使用注解来创建切面是 AspectJ 5 所引入的关键特性。AspectJ 5 之前，编写 AspectJ 切面需要学习 Java 语言的一种扩展，但是 AspectJ 面向注解的模型可以非常简便地通过少量注解把任意类转变为切面。

1. 定义切面

电视节目《最强大脑》是江苏卫视打造的一档益智类节目，收视率非常高，第四季最强大脑进行了改版，加入了一些创新元素，考虑到节目的观赏性和娱乐性，牺牲了节目的严谨性，观众褒贬不一。观众观看《最强大脑》的选手表演时，落座、关机、欣赏节目，表演精彩时鼓掌，不高兴时提前离开等；选手专注于表演，无需关心观众是否落座、鼓掌等行为，让我们应用切面编程来实现该需求。

我们已经定义了 Performance 接口，它是切面中切点的目标对象。现在，让我们使用 AspecJ 注解来定义切面。

一档好的电视节目如果没有观众，还有什么意义？观众很重要，但是对于最强大脑选手挑战项目来讲，它不是核心，是一个单独的关注点。因此我们把观众定义为一个切面。如下代码展示的 Audience 类定义了观众的功能。

为《最强大脑》节目定义观众，代码如下所示：

```
package com.ssm.chapter4.aop;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class Audience {
    @Before("execution( * com.ssm.chapter4.aop.Performance.perform(..))")
    public void takeSeat(){
        System.out.println("观众落座");
    }
    @Before("execution( * com.ssm.chapter4.aop.Performance.perform(..))")
    public void silenceMobile(){
        System.out.println("手机静音");
    }
    @AfterReturning("execution( * com.ssm.chapter4.aop.Performance.perform(..))")
    public void applaud(){
        System.out.println("鼓掌");
    }
```

```
    @AfterThrowing("execution( * com.ssm.chapter4.aop.Performance.perform(..))")
    public void demandRefund(){
        System.out.println("不，请把钱还给我！");
    }
}
```

Audience 类使用@AspectJ 注解进行了标注。该注解表明 Audience 不仅仅是一个 POJO，还是一个切面。Audience 类中的方法都使用注解来定义切面的具体行为。

Audience 有四个方法，定义了一个观众在观看演出时可能会做的事情。在演出之前，观众要就座(takeSeats())并将手机调至静音状态(silenceMobile())。如果演出很精彩，观众会鼓掌喝彩(applause())；不过，如果演出没有达到观众预期的话，观众会要求退款(demandRefund())。

可以看到，这些方法都使用了通知注解来表明它们应该在什么时候调用。AspectJ 提供了五个注解来定义通知，如表 4-2 所示。

表 4-2　Spring 使用 AspectJ 注解来声明通知方法

注　解	通　知
@after	通知方法会在目标方法返回或抛出异常后调用
@afterReturing	通知方法会在目标方法返回后调用
@ afterThrowing	通知方法会在目标方法抛出异常后调用
@ before	通知方法会将目标方法封装起来
@around	通知方法会在目标方法调用之前执行

Audience 使用到了表 4-2 中列出的三个通知，Audience 类使用这些注解声明通知，并为每个通知设置一个相同的切点表达式，显然它不是一种完美的设置切点的方法。如果我们定义一个切点，然后每次需要的时候引用它，那么这会是一个很好的方案。

我们可以这样做：@Pointcut 注解在一个@AspectJ 切面内定义可重用的切点。接下来的程序展现了新的 Audience，现在它使用了@Pointcut：

```
package com.ssm.chapter4.aop;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class Audience {
    @Pointcut("execution( * com.ssm.chapter4.aop.Performance.perform(..))")
    public void performance(){}
@Before("performance()")
    public void takeSeat(){
        System.out.println("观众落座");
```

```
    }
    @Before("performance()")
    public void silenceMobile(){
        System.out.println("手机静音");
    }
    @AfterReturning("performance()")
    public void applaud(){
        System.out.println("鼓掌");
    }
    @AfterThrowing("performance()")
    public void demandRefund(){
        System.out.println("不，请把钱还给我！");
    }
}
```

在 Audience 中，performance()方法使用了@Pointcut 注解。为@Pointcut 注解设置的值是一个切点表达式，就像之前在通知注解上所设置的那样。通过在 performance()方法上添加@Pointcut 注解，我们实际上扩展了切点表达式语言，我们现在把所有通知注解中的长表达式都替换成 performance()。performance()方法的实际内容并不重要，在这里它实际上应该是空的。其实该方法本身只是一个标识，供@Pointcut 注解依附。

需要注意的是，除了注解和没有实际操作的 performance()方法，Audience 类依然是一个 POJO。我们能够像使用其他的 Java 类那样调用它的方法，它的方法也能够独立地进行单元测试，这与其他的 Java 类并没有什么区别。Audience 只是一个 Java 类，只不过通过注解表明它可作为切面使用而已。

像其他的 Java 类一样，它可以装配为 Spring 中的 Bean：

```
    @Bean
    public Audience audience(){
        return new Audience();
    }
```

目前 Audience 对象只是 Spring 容器类普通的 Bean，虽然使用了 AspectJ 注解，但它还不是切面，这些注解不会被解析，如果使用 JavaConfig 的话，，可以在配置类的类级别上通过使用 EnableAspectJ-AutoProxy 注解启用自动代理功能。JavaConfig 开启 Aspect 注解的自动代理，代码如下所示：

```
package com.ssm.chapter4.aop;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
```

```
@Configuration
@EnableAspectJAutoProxy
@ComponentScan
public class JavaConfig {
    @Bean
    public Audience audience(){
        return new Audience();
    }
}
```

使用 JavaConfig 后，AspectJ 自动代理会为用@Aspect 注解的 Bean 创建一个代理，这个代理会封装切点所匹配的 Bean。在这种情况下，将会为 Audience Bean 创建一个代理，Audience 类中的通知方法将会在 perform()调用前后执行。我们一起测试 Audience 切面。

编写 Player 类实现 Performance 接口，代码如下所示：

```
package com.ssm.chapter4.aop;
public interface Performer {
    public void perform();
}
```

定义一个 Player 类实现 Performer 接口，代码如下所示：

```
package com.ssm.chapter4.aop;
public class Player implements Performance{
    private String name;//选手姓名
    private String subject;//比赛的题目(项目)
    @Override
    public void perform() {
        System.out.println(name+"正在挑战"+subject);
    }
}
```

编写测试类 AspectTest，代码如下所示：

```
package com.ssm.chapter4.test;
import static org.junit.Assert.fail;
import javax.annotation.Resource;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import com.ssm.chapter4.aop.JavaConfig;
import com.ssm.chapter4.aop.Performance;
@RunWith(SpringJUnit4ClassRunner.class)
```

```
@ContextConfiguration(classes = JavaConfig.class)
public class AspectTest {
    @Resource
    Performance player;
    @Test
    public void test() {
        player.perform();
    }
}
```

运行 Junit Test，测试结果如图 4-10 所示。

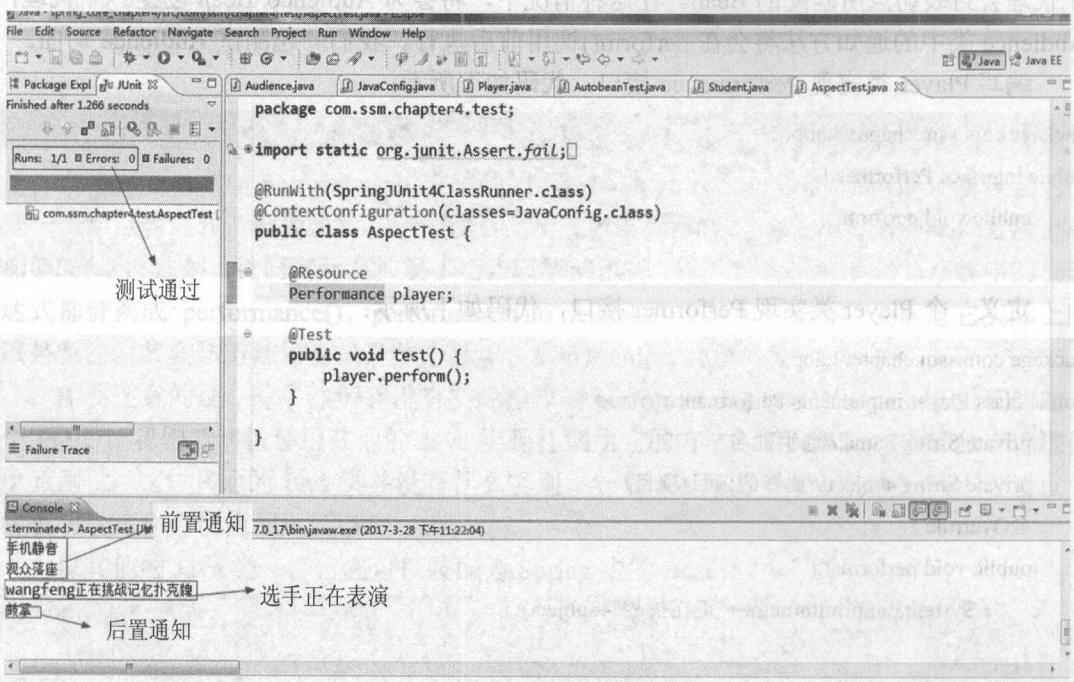

图 4-10　Aspect Test 测试结果

在 Spring 中，注解和自动代理提供了一种很便利的方式来创建切面。它非常简单，并且只涉及最少的 Spring 配置。但是，面向注解的切面声明有一个明显的劣势：你必须能够为通知类添加注解。为了做到这一点，必须要有源码。

如果你没有源码，或者不想将 AspectJ 注解放到你的代码之中，Spring 为切面提供了另外一种可选方案。下面介绍如何在 Spring XML 配置文件中声明切面。

4.4.4　在 XML 中声明切面

如果要在 XML 中配置切面，在 Spring 的 AOP 命名空间中提供了多个元素用于在 XML 中声明切面，如表 4-3 所示。

表 4-3　AOP 配置元素

AOP 配置元素	用　途
<aop:advisor>	定义 AOP 通知器
<aop:after>	定义 AOP 后置通知(不管被通知的方法是否执行成功)
<aop:after-returning>	定义 AOP 返回通知
<aop:after-throwing>	定义 AOP 异常通知
<aop:around>	定义 AOP 环绕通知
<aop:aspect>	定义一个切面
<aop:aspectj-autoproxy>	启用@Aspect 注解驱动的切面
<aop:before>	定义一个 AOP 前置通知
<aop:config>	顶层的 AOP 配置元素。大多数的<aop:*>元素必须包含在<aop:config>元素内
<aop:declare-parents>	以透明的方式为被通知的对象引入额外的接口
<aop:pointcut>	定义一个切点

为了使用 Spring 的 AOP 命名空间提供的声明切面的元素，将 Audience 类的注解移除，代码如下：

```
package com.ssm.chapter4.aop;
public class Audience {
    public void takeSeat(){
        System.out.println("观众落座");
    }
    public void silenceMobile(){
        System.out.println("手机静音");
    }
    public void applaud(){
        System.out.println("鼓掌");
    }
    public void demandRefund(){
        System.out.println("不，请把钱还给我！");
    }
}
```

上面定义的 Audience 类并没有任何特别之处，它就是有几个方法的简单 Java 类。我们可以像其他类一样把它注册为 Spring 应用上下文中的 Bean。尽管看起来并没有什么差别，但 Audience 已经具备了成为 AOP 通知的所有条件，我们可以利用 AOP 命名空间提供的元素将 Audience 类中的方法变成预期的通知。

(1) 声明前置通知和后置通知。

我们会使用 Spring AOP 命名空间中的一些元素，将没有注解的 Audience 转换为切面。下面的代码展示需要的 applicationContext.xml 元素：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
      xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop = "http://www.springframework.org/schema/aop"
      xmlns:context = "http://www.springframework.org/schema/context"
      xmlns:tx = "http://www.springframework.org/schema/tx"
      xsi:schemaLocation = "http://www.springframework.org/schema/beans
                            http://www.springframework.org/schema/beans/spring-beans.xsd
                            http://www.springframework.org/schema/aop
                            http://www.springframework.org/schema/aop/spring-aop.xsd
                            http://www.springframework.org/schema/context
                            http://www.springframework.org/schema/context/spring-context.xsd
                            http://www.springframework.org/schema/tx
                            http://www.springframework.org/schema/tx/spring-tx.xsd">
      <!-- <context:component-scan base-package = "com.ssm.chapter4"></context:component-scan>
      <bean name = "geLi" class = "com.ssm.chapter4.ioc.Actor"></bean>
      <bean name = "moAttack" class = "com.ssm.chapter4.ioc.MoAttack">
            <property name = "geLi" ref = "geLi"></property>
      </bean> -->
      <bean name = "audience" class = "com.ssm.chapter4.aop.Audience"></bean>
      <aop:config>
            <aop:aspect>
                  <aop:before method = "takeSeat"
                  pointcut = "execution( * com.ssm.chapter4.aop.Performance.perform(..))"/>
                  <aop:before method = "silenceMobile"
                  pointcut = "execution( * com.ssm.chapter4.aop.Performance.perform(..))"/>
                  <aop:after-returning method = "applaud"
                  pointcut = "execution( * com.ssm.chapter4.aop.Performance.perform(..))"/>
                  <aop:after-throwing method = "demandRefund"
                  pointcut = "execution( * com.ssm.chapter4.aop.Performance.perform(..))"/>
            </aop:aspect>
      </aop:config>
</beans>
```

关于 AOP 的配置元素的使用，应注意把它们放在<aop:config>元素中。该切面应用了四个不同的通知。两个<aop:before>元素定义了匹配切点的方法执行之前调用前置通知方法，也就是 Audience bean 的 takeSeats()和 turnOffCellPhones()方法(由 method 属性所声明)。

<aop:after-returning>元素定义了一个返回(after-returning)通知，在切点所匹配的方法调用之后再调用 applaud()方法。同样，<aop:after-throwing>元素定义了异常(after-throwing)通

知，如果所匹配的方法执行时抛出任何的异常，都将会调用 demandRefund()方法。

在基于 AspectJ 注解的通知中，当发现这种类型的重复时，我们使用@Pointcut 注解消除了这些重复的内容。而在基于 XML 的切面声明中，我们需要使用<aop:pointcut>元素。如下的 XML 展现了如何将通用的切点表达式抽取到一个切点声明中，这样这个声明就能在所有的通知元素中使用了。使用<aop:pointcut>定义切点代码如下所示：

```
<aop:config>
        <aop:aspect>
            <aop:pointcut expression = "execution( * com.ssm.chapter4.aop.Performance.perform(..))" id =
"performance"/>
            <aop:before method = "takeSeat" pointcut-ref = "performance"/>
            <aop:before method = "silenceMobile" pointcut-ref = "performance"/>
            <aop:after-returning method = "applaud" pointcut-ref = "performance"/>
            <aop:after-throwing method = "demandRefund" pointcut-ref = "performance"/>
        </aop:aspect>
    </aop:config>
```

(2) 声明环绕通知。

前置通知和后置通知的使用有一些局限。具体来说，如果不使用成员变量存储信息，在前置通知和后置通知之间共享信息会非常麻烦。

使用环绕通知，我们可以完成前置通知和后置通知所实现的相同功能，而且只需要在一个方法中实现。因为整个通知逻辑是在一个方法内实现的，所以不需要使用成员变量保存状态。修改 Audience 类，添加 watchPerform()，提供环绕通知。watchPerform 提供 AOP 的环绕通知，代码如下所示：

```
    public void watchPerform(ProceedingJoinPoint pjp){
        try {
            System.out.println("观众落座");
            System.out.println("手机静音");
            pjp.proceed();
            System.out.println("鼓掌");
        } catch (Throwable e) {
            System.out.println("不，请把钱还给我！");
        }
}
```

在观众切面中，watchPerformance()方法包含了之前四个通知方法的所有功能。不过，所有的功能都放在了这一个方法中，因此这个方法还要负责自身的异常处理。声明环绕通知与声明其他类型的通知并没有太大区别。我们所需要做的仅仅是使用<aop:around>元素。在 XML 中使用<aop:around>声明环绕通知，代码如下所示：

```
<aop:config>
        <aop:aspect>
```

```
            <aop:pointcut expression = "execution( * com.ssm.chapter4.aop.Performance.perform(..))" id =
"performance"/>
            <aop:around method = "watchPerform" pointcut-ref = "performance"/>
        </aop:aspect>
</aop:config>
```

像其他通知的 XML 元素一样，<aop:around>指定了一个切点和一个通知方法的名字，将 performance 对象的 watchPerform 方法变成了环绕通知。

4.5 Spring 的事务管理

理解事务之前，先举一个日常生活常发生的事情：取钱。比如你去 ATM 机取 1000 元钱，大体有两个步骤：首先输入密码和想要取出的金额，银行卡扣掉 1000 元钱；然后 ATM 机吐出 1000 元钱。这两个动作序列必须是要么都执行，要么都不执行。如果银行卡扣除了 1000 元钱但是 ATM 机出钱失败，你将会损失 1000 元钱；如果银行卡扣钱失败但是 ATM 机却吐出了 1000 元钱，那么银行将损失 1000 元钱。所以，如果一个动作成功另一个动作失败，整个过程就是失败的。整个取钱过程应都能回滚，也就是完全取消所有操作的话，这个结果对双方是可以接受的。

事务就是用来解决类似问题的。事务是一系列的动作，这些动作必须全部完成，如果有一个失败的话，那么事务就会回滚到最开始的状态，仿佛什么都没发生过一样。在企业级应用程序开发中，事务管理是必不可少的技术，用来确保数据的完整性和一致性。

4.5.1 事务的特性

事务有四大特性，简称 ACID。

原子性(Atomicity)：事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用。

一致性(Consistency)：一旦事务完成(不管成功还是失败)，系统必须确保它所建模的业务处于一致的状态，而不会是部分完成部分失败。现实中的数据不应该被破坏。

隔离性(Isolation)：可能有许多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。

持久性(Durability)：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

4.5.2 核心接口

Spring 事务管理的实现有许多细节，如果对整个接口框架有个大体了解会非常有利于我们理解事务，Spring 事务接口框架如图 4-11 所示。

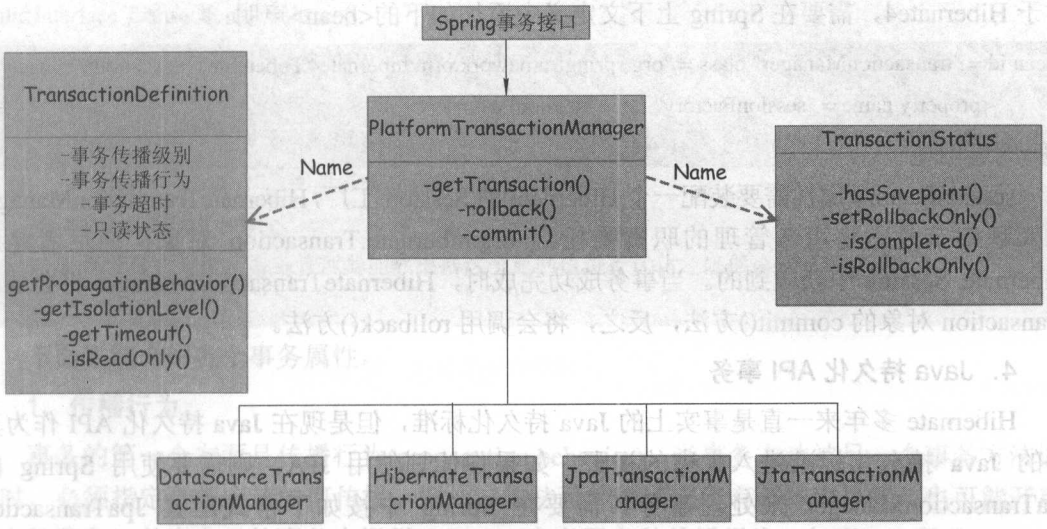

图 4-11　Spring 事务接口框架

下面通过讲解 Spring 的事务接口来了解 Spring 实现事务的具体策略。

1. 事务管理器

Spring 并不直接管理事务，而是提供了多种事务管理器，它们将事务管理的职责委托 Hibernate 或者 JTA 等持久化机制所提供的相关平台框架的事务来实现。

Spring 事务管理器的接口是 org.springframework.transaction.PlatformTransactionManager，通过这个接口，Spring 为各个平台如 JDBC、Hibernate 等都提供对应的事务管理器，但是具体的实现就是各个平台自己的事情了。此接口的内容如下：

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}
```

2. JDBC 事务

如果应用程序中直接使用 JDBC 来进行持久化，DataSourceTransactionManager 会为你处理事务边界。为了使用 DataSourceTransactionManager，需要使用如下的 XML 将其装配到应用程序的上下文定义中：

```
<bean id = "transactionManager" class = "org.springframework.jdbc.datasource.
                                        DataSourceTransactionManager">
    <property name = "dataSource" ref = "dataSource" />
</bean>
```

3. Hibernate 事务

如果应用程序的持久化是通过 Hibernate 实现的，则需要使用 HibernateTransactionManager。

对于 Hibernate4，需要在 Spring 上下文定义中添加如下的<bean>声明：

```
<bean id = "transactionManager" class = "org.springframework.orm.hibernate4.HibernateTransactionManager">
        <property name = "sessionFactory" ref = "sessionFactory" />
</bean>
```

sessionFactory 属性需要装配一个 Hibernate 的 Session 工厂，HibernateTransactionManager 的实现细节是它将事务管理的职责委托给 org.hibernate.Transaction 对象，而后者是从 Hibernate Session 中获取到的。当事务成功完成时，HibernateTransactionManager 将会调用 Transaction 对象的 commit()方法，反之，将会调用 rollback()方法。

4. Java 持久化 API 事务

Hibernate 多年来一直是事实上的 Java 持久化标准，但是现在 Java 持久化 API 作为真正的 Java 持久化标准进入大家的视野。如果你计划使用 JPA，则需要使用 Spring 的 JpaTransactionManager 来处理事务。需要在 Spring 中按如下格式配置 JpaTransaction Manager：

```
<bean id = "transactionManager" class = "org.springframework.orm.jpa.JpaTransactionManager">
        <property name = "sessionFactory" ref = "sessionFactory" />
</bean>
```

JpaTransactionManager 只需要装配一个 JPA 实体管理工厂(javax.persistence.Entity ManagerFactory 接口的任意实现)。JpaTransactionManager 将与由工厂所产生的 JPA Entity Manager 合作来构建事务。

5. Java 原生 API 事务

如果你没有使用以上所述的事务管理，或者是跨越了多个事务管理源(比如两个或者是多个不同的数据源)，就需要使用 JtaTransactionManager，代码如下所示：

```
<bean id = "transactionManager" class = "org.springframework.transaction.jta.JtaTransactionManager">
        <property name = "transactionManagerName" value = "java:/TransactionManager" />
    </bean>
```

JtaTransactionManager 将事务管理的责任委托给 javax.transaction.UserTransaction 和 javax.transaction.TransactionManager 对象，其中事务成功完成时通过 UserTransaction. commit()方法提交，事务失败时通过 UserTransaction.rollback()方法回滚。

4.5.3 基本事务属性

事务管理器接口 PlatformTransactionManager 通过 getTransaction(TransactionDefinition definition)方法来得到事务，这个方法里面的参数是 TransactionDefinition 类，这个类就定义了一些基本的事务属性。那么什么是事务属性呢？事务属性可以理解成事务的一些基本配置，描述了事务策略如何应用到方法上。事务属性包含了 5 个方面，分别是传播行为、隔离规则、回滚规则、事务超时和是否可读。

TransactionDefinition 接口的内容如下：

```
public interface TransactionDefinition {
    int getPropagationBehavior(); // 返回事务的传播行为
    int getIsolationLevel();
     // 返回事务的隔离级别，事务管理器根据它来控制另外一个事务可以看到本事务内的哪些数据
    int getTimeout();  // 返回事务必须在多少秒内完成
    boolean isReadOnly();
    // 事务是否只读，事务管理器能够根据这个返回值进行优化，确保事务是只读的
}
```

下面详细介绍各个事务属性。

1. 传播行为

事务的第一个方面是传播行为(propagation behavior)。当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。Spring 定义了七种传播行为，如表 4-4 所示。

表 4-4　Spring 事务传播行为

传播行为	含　义
PROPAGATION_REQUIRED	表示当前方法必须运行在事务中。如果当前事务存在，方法将会在该事务中运行。否则，会启动一个新的事务
PROPAGATION_SUPPORTS	表示当前方法不需要事务上下文，但是如果存在当前事务，则该方法会在这个事务中运行
PROPAGATION_MANDATORY	表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常
PROPAGATION_REQUIRED_NEW	表示当前方法必须运行在它自己的事务中。一个新的事务将被启动。如果存在当前事务，在该方法执行期间，当前事务会被挂起。如果使用 JTATransactionManager，则需要访问 TransactionManager
PROPAGATION_NOT_SUPPORTED	表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用 JTATransaction Manager，则需要访问 TransactionManager
PROPAGATION_NEVER	表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常
PROPAGATION_NESTED	表示如果当前已经存在一个事务，则该方法将会在嵌套事务中运行。嵌套事务可以独立于当前事务进行单独提交或回滚。如果当前事务不存在，那么其行为与 PROPAGATION_REQUIRED 一样。注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确认它们是否支持嵌套事务

2. 隔离级别

事务的第二个维度就是隔离级别(isolation level)。隔离级别定义了一个事务可能受其他并发事务影响的程度。在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务。隔离级别如表 4-5 所示。

表 4-5 隔 离 级 别

隔 离 级 别	含 义
ISOLATION_DEFAULT	使用后端数据库默认的隔离级别
ISOLATION_READ_UNCOMMITTED	最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读(见下文解释)
ISOLATION_READ_COMMITTED	允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
ISOLATION_REPEATABLE_READ	对同一字段的多次读取结果都是一致的，除非数据是被本身事务所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生
ISOLATION_SERIALIZABLE	最高的隔离级别，完全服从 ACID 的隔离级别，确保阻止脏读、不可重复读以及幻读，也是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数据库表来实现

并发虽然是必需的，但可能会导致以下的问题：

① 脏读(Dirty reads)——脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。

② 不可重复读(Nonrepeatable read)——不可重复读发生在一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据时。这通常是因为另一个并发事务在两次查询期间进行了更新。

③ 幻读(Phantom read)——幻读与不可重复读类似。它发生在一个事务(T1)读取了几行数据，接着另一个并发事务(T2)插入了一些数据时。在随后的查询中，第一个事务(T1)就会发现多了一些原本不存在的记录。

3. 只读

事务的第三个特性是它是否为只读事务。如果事务只对后端的数据库进行该操作，数据库可以利用事务的只读特性来进行一些特定的优化。通过将事务设置为只读，就可以给数据库一个机会，让它应用它认为合适的优化措施。

4. 事务超时

为了使应用程序很好地运行，事务不能运行太长的时间。因为事务可能涉及对后端数据库的锁定，所以长时间的事务运行会不必要地占用数据库资源。设置事务超时属性值，当事务执行时间超过属性值时，事物如果没有执行完毕，就会自动回滚。

5. 回滚规则

事务属性定义了哪些异常会导致事务回滚而哪些不会。默认情况下，事务只有遇到运行期异常时才会回滚，而在遇到检查型异常时不会回滚(这一行为与 EJB 的回滚行为是一致的)。但是你可以声明事务在遇到特定的检查型异常时像遇到运行期异常那样回滚。同样，你还可以声明事务遇到特定的异常不回滚，即使这些异常是运行期异常。

4.5.3 事务状态

上面讲到的调用 PlatformTransactionManager 接口的 getTransaction()的方法得到的是 TransactionStatus 接口的一个实现，这个接口的内容如下：

```
public interface TransactionStatus{
        boolean isNewTransaction(); // 是否是新的事物
        boolean hasSavepoint(); // 是否有恢复点
        void setRollbackOnly();  // 设置为只回滚
        boolean isRollbackOnly(); // 是否为只回滚
        boolean isCompleted; // 是否已完成
}
```

可以发现这个接口描述的是一些处理事务、提供简单的控制事务执行和查询事务状态的方法，在回滚或提交的时候需要应用对应的事务状态。

4.5.4 声明事务管理实例

(1) 创建事务处理的接口 FooService 以及 Foo 类。

代码如下：

```
package com.ssm.chapter4.service;
import com.ssm.chapter4.vo.Foo;
public interface FooService {
    Foo getFoo(String fooName);
    Foo getFoo(String fooName, String barName);
    void insertFoo(Foo foo);
    void updateFoo(Foo foo);
}
package com.ssm.chapter4.vo;
public class Foo{
}
```

(2) 创建 FooService 的实现类 DefaultFooService。

代码如下：

```
package com.ssm.chapter4.service;
import com.ssm.chapter4.vo.Foo;
```

```
public class DefaultFooService implements FooService{
        public Foo getFoo(String fooName) {
            throw new UnsupportedOperationException();
        }
        public Foo getFoo(String fooName, String barName) {
            throw new UnsupportedOperationException();
        }
        public void insertFoo(Foo foo) {
            throw new UnsupportedOperationException();
        }
        public void updateFoo(Foo foo) {
            throw new UnsupportedOperationException();
        }
          public void deleteFoo(Foo foo) {
            throw new UnsupportedOperationException();
        }
}
```

假设 FooService 接口里的方法接受容器的事务管理，其中 getFoo(String fooName)和 getFoo(String fooName, String barName)以只读方法执行事务，insertFoo(Foo foo)，updateFoo(Foo foo)和 deleteFoo(Foo foo)以读写方式执行事务。在 applicationContext.xml 配置代码如下：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
      xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop = "http://www.springframework.org/schema/aop"
      xmlns:context = "http://www.springframework.org/schema/context"
      xmlns:tx = "http://www.springframework.org/schema/tx"
      xsi:schemaLocation = "http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
                                  http://www.springframework.org/schema/aop
                                  http://www.springframework.org/schema/aop/spring-aop.xsd
                                  http://www.springframework.org/schema/context
                                  http://www.springframework.org/schema/context/spring-context.xsd
                                  http://www.springframework.org/schema/tx
                                  http://www.springframework.org/schema/tx/spring-tx.xsd">

      <!-- 声明一个事务管理的fooService bean -->
<bean id = "fooService" class = "com.ssm.chapter4.service.DefaultFooService"></bean>
```

```
    <!-- 声明事务通知 -->
    <tx:advice id = "txAdvice" transaction-manager = "txManager">
        <!--属性-->
        <tx:attributes>
            <!-- 所有以get开头的方法 read-only -->
            <tx:method name = "get*" propagation = "SUPPORTS" read-only = "true"/>
            <!-- 其他方法使用事务读写设置 -->
            <tx:method name = "insert*" propagation = "REQUIRED" read-only = "false"/>
<tx:method name = "update*" propagation = "REQUIRED" read-only = "false"/>
<tx:method name = "delete*" propagation = "REQUIRED" read-only = "false"/>
        </tx:attributes>
    </tx:advice>
    <!-- 声明事务切面 -->
     <aop:config>
        <aop:pointcut id = "fooServiceOperation" expression = "execution(* com.ssm.chapter4.service.
                                FooService.*(..))"/>
<aop:advisor advice-ref = "txAdvice" pointcut-ref = "fooServiceOperation"/>
    </aop:config>
    <!-- 声明数据源bean -->
<bean id = "dataSource" class = "org.apache.commons.dbcp2.BasicDataSource" destroy-method = "close">
        <property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>
        <property name = "url" value = "jdbc:mysql:///mytest"/>
        <property name = "username" value = "root"/>
        <property name = "password" value = "root"/>
    </bean>
    <!-- 声明一个事务管理器 -->
    <bean id = "txManager" class = "org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name = "dataSource" ref = "dataSource"/>
     </bean>
</beans>
```

上面的代码配置好了事务切面，在调用 FooService 的 get 开头的方法时置入事务通知。

本章小结

本章描述了 Spring 的核心知识，包括 Bean 的装配及 Spring 依赖注入，主要有构造方法注入、属性注入和接口注入；面向切面编程，主要用两种方式来声明切面，一种以注解的方法将普通类声明为切面，另一种是通过 XML 的方法来声明切面。

练 习 题

一、选择题

1. 下面关于 Spring 的说法正确的是(　　)。

A. Spring 是一个重量级的框架　　B. Spring 是一个轻量级的框架

C. Spring 是一个 IoC 和 AOP 容器　　D. Spring 是一个入侵式的框架

2. 下面关于 IoC 的理解，正确的是(　　)。

A. 控制反转　　B. 对象被动地接受依赖类

C. 对象主动地去寻找依赖类　　D. 一定要用接口

3. 下面关于 AOP 的理解，正确的是(　　)。

A. 面向纵向的开发　　B. 面向横向的开发

C. AOP 关注的是面　　D. AOP 关注的是点

4. Spring 一共由(　　)块组成。

A. 1　　B. 3　　C. 5　　D. 7

5. 下列关于 Spring 各模块之间关系的叙述，正确的是(　　)。

A. Spring 各模块之间是紧密联系的、相互依赖的

B. Spring 各模块之间可以单独存在

C. Spring 的核心模块是必需的，其他模块是基于核心模块的

D. Spring 的核心模块不是必需的，可以不要

6. Spring 核心模块的作用是(　　)。

A. 做 AOP 的　　B. 做 IoC 的，用来管理 Bean

C. 用来支持 Hibernete　　D. 用来支持 Struts

7. Spring 的通知类型有(　　)。

A. Before 通知　　B. After return 通知

C. Throws 通知　　D. Around 通知

8. 下面关于切入点的说法正确的是(　　)。

A. 是 AOP 中一系列连接点的集合

B. 在做 AOP 时定义切入点是必需的

C. 在做 AOP 时定义切入点不是必需的

D. 可以用正则表达式来定义切入点

9. 下面关于 Spring 依赖注入方式正确的是(　　)。

A. set 方法注入　　B. 构造方法注入

C. get 方法注入　　D. 接口的注入

10. 下面关于在 Spring 中配置 Bean 的 id 属性的说法正确的是(　　)。

A. id 属性是必需的，没有 id 属性就会报错

B. id 属性不是必需的，可以没有

C．id 属性的值可以重复
D．id 属性的值不可以重复
11．下面关于在 Spring 中配置 Bean 的 name 属性的说法正确的是(　　)。
A．name 属性是必需的，没有 name 属性就会报错
B．name 属性不是必需的，可以没有
C．name 属性的值可以重复
D．name 属性的值不可以重复
12．下面(　　)是 IoC 自动装载方法。
A．byName　　B．byType　　C．constructor　　D．byMethod
13．下面关于在 Spring 中配置 Bean 的 init-method 的说法正确的是(　　)。
A．init-method 是在最前面执行的
B．init-method 在构造方法后、依赖注入前执行
C．init-method 在依赖注入之后执行
D．init-method 在依赖注入之后、构造函数之前执行
14．下面关于 Spring 配置文件的说法正确的是(　　)
A．Spring 配置文件必须叫 applicationContext.xml
B．Spring 配置文件可以不叫 applicationContext.xml
C．Spring 配置文件可以有多个
D．Spring 配置文件只能有一个
15．下面关于构造注入优点的说法错误的是(　　)。
A．构造期即创建一个完整、合法的对象
B．不需要写繁琐的 setter 方法
C．对于复杂的依赖关系，构造注入更简洁、直观
D．在构造函数中决定依赖关系的注入顺序
16．下面关于 AOP 的理解正确的是(　　)。
A．能够降低组件之间的依赖关系
B．将项目中的公共的问题集中解决，减少代码量，提高系统的可维护性
C．AOP 是面向对象的代替品
D．AOP 不是面向对象的代替品，是面向对象很好的补充
17．下面关于 Spring 框架的说明中错误的是(　　)。
A．可以作用于任何 Java 应用
B．Spring 是侵入式的
C．基于 Spring 开发的应用中的对象一般不依赖于 Spring 的类
D．Spring 是一个容器，因为它包含并且管理应用 JavaBean 对象的生命周期和配置
18．以下(　　)不是 Spring 中 DI 的注入方式。
A．接口注入　　B．getter/setter 注入
C．构造器注入　　D．对象注入
19．下列关于 Spring 框架的说明中正确的是(　　)。
A．只能作用于任何 JavaWeb 应用

B．Spring 是侵入式的

C．基于 Spring 开发的应用中的对象需要依赖于 Spring 的类

D．Spring 是一个容器，因为它包含并且管理应用 JavaBean 对象的生命周期和配置

20. 以下关于 Spring 中 DI 的注入方式的选择正确的是(　　)。

(1) 接口注入

(2) getter/setter 注入

A．都是　　B．只有 1 是　　C．只有 2 是　　D．都不是

21．在 SSM 框架中，Spring 承担的责任是(　　)。

A．定义实体类　　B．数据的增删改查操作

C．业务逻辑的描述　　D．页面展示和控制转发

二、简答题

1. Spring 中的 AOP 是什么？举例说明。

2. 简述 AOP 和 OOP 的区别？

第五章 SpringMVC

本章讨论 SpringMVC(MVC-模型-视图-控制器)。MVC 是一个众所周知的以设计界面应用程序为基础的设计模式。它主要通过分离模型、视图及控制器在应用程序中的角色，从而将业务逻辑从界面中解耦。通常，模型负责封装应用程序数据以便其在视图层展示。视图仅仅负责展示这些数据，不包含任何业务逻辑。控制器则接收来自用户的请求，并调用后台服务来处理业务逻辑。处理后，后台业务层可能会返回一些需要在视图层展示的数据。控制器收集这些数据及准备模型在视图层展示。MVC 模式的核心思想是将业务逻辑从界面中分离出来，允许它们单独改变而不会相互影响。本章我们将对 SpringMVC 进行学习。

本章知识要点

- SpringMVC 概述；
- 创建第一个 SpringMVC 程序；
- SpringMVC RequestMapping 的基本设置；
- SpringMVC 前后台数据交互；
- SpringMVC 文件上传下载；
- SpringMVC 常用注解。

5.1 SpringMVC 概述

SpringMVC 是 SpringFrameWork 的后续产品，并已经融合在 Spring Web Flow 中。Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。使用 Spring 可插入 MVC 架构，可以选择使用内置的 Spring Web 框架或类似 Struts 这样的 Web 框架。通过策略接口，Spring 框架是高度可配置的，而且包含多种视图技术，例如 JavaServer Pages(JSP)、Velocity、Tiles、iText、POI 等技术。SpringMVC 框架并不知道使用的视图，所以不会要求只使用 JSP 技术。SpringMVC 分离了控制器、模型对象、分派器以及处理程序对象的角色，这种分离让它们更容易进行定制。在最简单的 SpringMVC 应用程序中，控制器是唯一的需要在 Java Web 部署描述文件(即 web.xml 文件)中配置的 Servlet。SpringMVC 控制器——通常称作 Dispatcher Servlet，实现了前端控制器设计模式，并且每个 Web 请求必须通过它以便它能够管理整个请求的生命周期。

当一个 Web 请求发送到 SpringMVC 应用程序，dispatcher servlet 首先接收请求。然后组织那些在 Spring Web 应用程序上下文配置的(例如实际请求处理控制器和视图解析器)或者使用注解配置的组件，所有的这些都需要处理该请求。如图 5-1 所示为 SpringMVC 处理请求的过程。

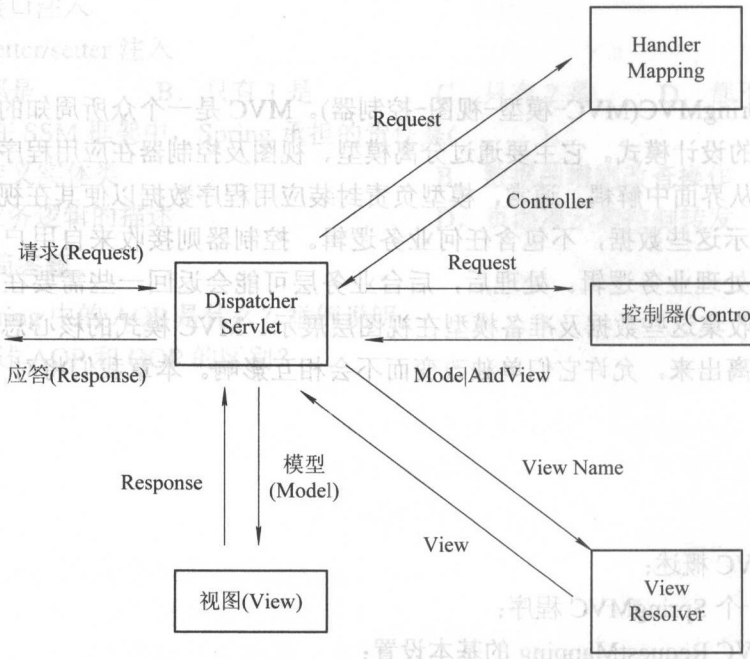

图 5-1　SpringMVC 处理请求

在 Spring 中定义一个控制器类，这个类必须标有@Controller 注解。当有@Controller 注解的控制器收到一个请求时，它会寻找一个合适的 Handler 方法去处理这个请求。这就需要控制器通过一个或多个 Handler 映射去把每个请求映射到 Handler 方法。为了这样做，一个控制器类的方法需要被@RequestMapping 注解装饰，使它们成为 Handler 方法。

Handler 方法处理完请求后，将把控制权委托给视图名与 Handler 方法返回值相同的视图。为了提供一个灵活的方法，一个 Handler 方法的返回值并不代表一个视图的实现而是一个逻辑视图，即没有任何文件扩展名。可以将这些逻辑视图映射到正确的实现，并将这些实现写入到上下文文件中，这样就可以轻松地更改视图层代码甚至不用修改请求 Handler 类的代码。为一个逻辑名称匹配正确的文件是视图解析器的责任。一旦控制器类已将一个视图名称解析到一个视图实现，它会根据视图实现的设计来渲染对应对象。下面一节中将创建第一个 SpringMVC 程序。

5.2 创建第一个 SpringMVC 程序

本节将通过建立一个简单的 SpringMVC 程序帮助读者理解 SpringMVC 程序的开发步骤。

5.2.1 新建项目

本文并不限定使用什么类型的 IDE(如 Eclipse、NetBeans IDE，或者 IntelliJ IDEA，它们通过提供自动完成、重构、调试特性在很大程度上简化了开发)来编码，可以选择 IDE。本书中 IDE 采用 Eclipse。在 Eclipse 里创建一个 Java Web 项目，名为 springmvc-demo。

5.2.2 导入 jar 包

对于一个 SpringMVC 的程序只需将以下的 jar 包添加到项目的 WEB-INF/lib 中即可。其中 xxx 代表 jar 的版本号，根据实际情况可以选择不同的版本号。

commons-logging-xxx.jar
spring-aop-xxx.jar
spring-beans-xxx.jar
spring-context-xxx.jar
spring-core-xxx.jar
spring-expression-xxx.jar
spring-web-xxx.jar
spring-webmvc-xxx.jar

如果项目正在使用 maven，那么配置这些 jar 包依赖就变得简单多了。在 pom.xml 中添加以下依赖即可：

```
<!-- Spring 版本号 -->
<properties>
        <spring.version>xxx</spring.version>
</properties>
<!-- Spring 依赖 -->
<dependencies>
<!-- Spring 核心包 -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-core</artifactId>
<version>${spring.version}</version>
</dependency>
<!-- Spring Web支持包 -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-web</artifactId>
<version>${spring.version}</version>
</dependency>
<!-- SpringMVC支持包 -->
```

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-webmvc</artifactId>
<version>${spring.version}</version>
</dependency>
</dependencies>
```

5.2.3 在 web.xml 中添加 SpringMVC 的配置

```
<servlet>
        <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
<!-- DispatcherServlet 初始化配置 -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:spring-mvc.xml</param-value>
        </init-param>
<!-- 是否在启动的时候就加载 -->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

这里我们将 DispatcherServlet 命名为 springmvc，在 servlet 的配置中，<load-on-startup>1</load-on-startup>的含义是标记容器是否在启动的时候就加载这个 servlet。当值为 0 或者大于 0 时，表示容器在应用启动时就加载这个 servlet；当是一个负数时或者没有指定时，则指示容器在该 servlet 被选择时才加载。正数的值越小，启动该 servlet 的优先级越高。这里值为 1，因此在 Web 项目一启动时就加载。接下来它会在类路径下加载名称为 spring-mvc.xml 的配置文件。这个文件中可以定义各种各样的 SpringMVC 需要使用的 Bean。需要说明的是，对于整个 Web 项目里的 Spring 配置文件中定义的 Bean，在这个配置文件中是可以继承的。上面我们将所有的请求都交给 DispatcherServlet。

5.2.4 在类路径下添加 SpringMVC 的配置

SpringMVC 的配置文件名为 spring-mvc.xml，其中内容如下：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context = "http://www.springframework.org/schema/context"
```

```
    xmlns:mvc = "http://www.springframework.org/schema/mvc"
    xsi:schemaLocation                    =                    "http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd
        http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd">
    <!-- 启动spring自动扫描 -->
    <context:component-scan base-package = "com.test.controller"/>
   <!-- 启动SpringMVC的注解功能，完成请求和注解POJO的映射 -->
 <mvc:annotation-driven />
        <!-- 配置视图解析器 -->
    <bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver"
            id = "internalResourceViewResolver">
        <!-- 前缀 -->
        <property name = "prefix" value = "/WEB-INF/jsp/" />
        <!-- 后缀 -->
        <property name = "suffix" value = ".jsp" />
    </bean>
</beans>
```

SpringMVC 在 Controller 控制层方法中通常返回的是逻辑视图，如何定位到真正的页面，就需要通过视图解析器。SpringMVC 里提供了多个视图解析器，InternalResourceViewResolver 就是其中之一，InternalResourceViewResolver 是比较常用的视图解析器，InternalResourceViewResolver 解析器会自动添加前缀(prefix)和后缀(suffix)，在上面的配置中当处理器 Controller 中的方式返回逻辑视图字符串“index”时，它实际要定向的页面应该是/WEB-INF/jsp/index.jsp。

5.2.5 建立视图文件

在 WEB-INF/jsp 下建立 ShowUser.jsp，用于显示测试成功的页面，其代码如下所示：

```
<%@ page language = "java" pageEncoding = "utf-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>测试</title>
</head>
<body>
    测试页面！
</body>
</html>
```

5.2.6 建立 Controller 控制层文件

在 com.test.controller 包下建立 UserController.java，控制器主要负责从视图读取数据，控制用户输入，并向模型发送数据，然后把模型处理的结果返回给视图。代码如下所示：

```
package com.test.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping("/showUser")
    public String showUser() {
        return "ShowUser";
    }
}
```

上面的 showUser 方法执行完成后返回“ShowUser”字符串，这个字符串是一个逻辑视图，它对应一个具体的视图，根据 5.2.4 小节关于 spring-mvc.xml 中 ViewResolver 的配置，我们可以知道它对应的具体视图是/WEB-INF/jsp/ShowUser.jsp。常见的逻辑视图返回方式除了直接以字符形式返回，SpringMVC 常常还通过 ModelMap 或 ModelAndView 方式返回逻辑视图。如 ModelAndView 方式返回逻辑视图的代码如下所示：

```
@RequestMapping("/showUser")
    public ModelAndView showUser() {
        return new ModelAndView("ShowUser", "message", "test message!");
    }
```

第一个参数是逻辑视图字符串，第二个参数是要往 ShowUser 视图上传递参数的名称，第三个参数是要往 ShowUser 视图上传递参数的值。

5.2.7 部署运行项目

项目部署好之后，启动服务器，在浏览器里输入如下地址进行测试：http://localhost:8080/springmvc-demo/user/showUser，可以看到如图 5-2 所示结果。

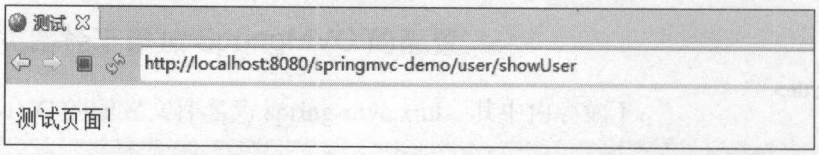

图 5-2 测试结果

当能看到上面的页面说明时我们的第一个 SpringMVC 程序就已经正确地建立好了。本程序在 Eclipse 里的结构如图 5-3 所示。

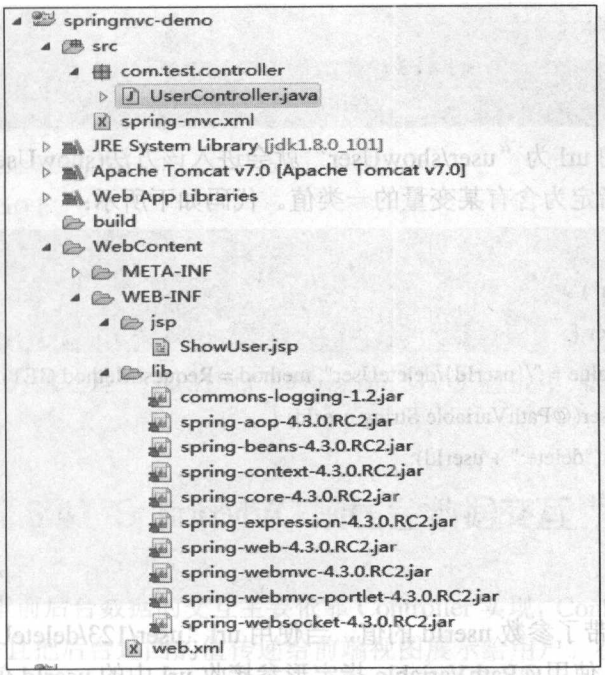

图 5-3 程序在 Eclipse 里的结构

上面讲解了 SpringMVC 第一个程序的建立，接下来我们将分别介绍 SpringMVC 在开发中的常用配置用法。

5.3 SpringMVC RequestMapping 的基本设置

在上一节中类的上面注解@RequestMapping("/user")表示所有的 user 请求会全部进入该类进行处理，同时也在 showUser 方法上加上了@RequestMapping("/showUser")，表示所有的 showUser 请求都会进入该 Controller。

在 SpringMVC 中自定义的 controller 中会调用有@RequestMapping 注解字样的方法来处理请求。当然可以编写多个处理请求的方法，而这些方法的调用都是通过@RequestMapping 的属性类控制调用的。

(1) @RequestMapping 的 value 属性需要指定请求的实际地址，指定的地址可以是 URI Template 模式(最终请求的 url 为类的注解的 url+方法注解的 url)，@RequestMapping 的 value 属性值有以下三类：

第一类：可以指定为普通的具体值。代码如下所示：

```
@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping("/showUser")
    public String showUser() {
```

```
        return "ShowUser";
    }
}
```

该注解当请求的 url 为“user/showUser”就会进入该方法(showUser)处理。

第二类：可以指定为含有某变量的一类值。代码如下所示：

```
@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping(value = "/{userId}/deleteUser", method = RequestMethod.GET)
    public String deleteUser(@PathVariable String userId) {
        System.out.println("delete:" + userId);
        return "ShowUser";
    }
}
```

这个注解 url 中带了参数 userId 的值，当使用 url“user/123/deleteUser”时“123”会被匹配成 userId 的值，使用@PathVariable 指定形参接收 url 中的 userId 值。

第三类：可以指定为含正则表达式的一类值。代码如下所示：

```
@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping(value = "/{userBirth:\\d{4}-\\d{2}-\\d{2}}/updateUser")
    public String updateUser(@PathVariable String userBirth) {
        System.out.println("userBirth:" + userBirth);
        return "ShowUser";
    }
}
```

这个注解当请求的 url 类似 user/1992-09-18/updateUser 时，userBirth 的值通过“\\d{4}-\\d{2}-\\d{2}”正则表达式来匹配，使用@PathVariable 指定形参接收 url 中的 userBirth 的值。

(2) method：指定请求的 method 类型，如 GET、POST、PUT、DELETE 等(也就是说只有制定类型的请求才会进入该方法处理)。

(3) consumes：指定处理请求的提交内容类型(Content-Type)，例如 application/json、text/html。

(4) produces：指定返回的内容类型，仅当 request 请求头中的 Accept 类型中包含该指定类型才返回。

(5) params：指定 request 中必须包含某些参数值时才让该方法处理请求。

(6) headers：指定 request 中必须包含某些指定的 header 值时才让该方法处理请求。

当类没有@RequestMapping 注解时，则直接参考方法的注解匹配对应的 url。代码

如下：

```
@Controller
public class UserController {
    @RequestMapping("/showUser")
    public String showUser() {
        return "ShowUser";
    }
}
```

这里当我们输入的地址是“showUser”时 SpringMVC 直接用 showUser 方法来处理请求。

5.4 SpringMVC 前后台数据交互

在 SpringMVC 里前后台数据的交互主要依靠 Controller 实现，Controller 获取前端视图向后台请求的值，并且把后台返回的值传递给前端视图展示给用户。在上一节中类的上面加上注解@Controller，表示该类是一个处理请求和应答的 Controller，接下来我们将学习 Controller 如何实现前后台数据交互。

5.4.1 Controller 获取前台传递的参数

以下是写在 HTML 表单里待传递的前台数据代码片段：

```
<form action = "addUser" method = "post">
    用户名:<input type = "text" name = "name"/><br/>
    年龄:<input type = "text" name = "age"/><br/>
    <input type = "submit" value = "添加"/>
</form>
```

接收前台表单数据通常可以使用以下两种方式。

方式一：直接使用形参获取前台传递的参数，要注意的是形参的名字必须和页面参数的名字一致。代码如下所示：

```
@RequestMapping(value = "/addUser", method = RequestMethod.POST)
    public String addUser(Model model, String name, Integer age) {
        System.out.println("name:" + name + "age:" + age);
        return "ShowUser";
    }
```

如果上面的形参和页面参数的名字不一致则可以使用@ModelAttribute 来指定形参要接收的参数的值。上面页面参数的名字 name 的值可以用形参名为 nickname 的来接收它传递上来的值，代码如下所示：

```
@RequestMapping(value = "/addUser", method = RequestMethod.POST)
```

```
    public String addUser(Model model, @ModelAttribute("name")String nickname, Integer age) {
        System.out.println("name:" + nickname + "age:" + age);
        return "ShowUser";
    }
```

方式二：使用对象接受前台传递的参数，要注意的是前台传递的参数的名称必须和对象的属性名称一致，代码如下所示：

```
@RequestMapping(value = "/addUser", method = RequestMethod.POST)
  public String addUser(Model model, User user) {
    System.out.println("name:" + user.getName() + "age:" + user.getAge());
    return "ShowUser";
  }
```

如果上面的形参和页面参数的名字不一致也可以使用@ModelAttribute 的方式来指定要接收的参数值，代码如下所示：

```
@RequestMapping(value = "/addUser", method = RequestMethod.POST)
  public String addUser(Model model, User user, @ModelAttribute("name")String nickname) {
    System.out.println("name:" + nickname + "age:" + user.getAge());
    return "ShowUser";
  }
```

上面在方法的入参前使用@ModelAttribute 注解，页面参数的名字 name 的值可以用形参名为 nickname 来接收，当在方法定义上使用@ModelAttribute 注解时，SpringMVC 在调用目标处理方法前，会先逐个调用在方法级上标注了@ModelAttribute 的方法，代码如下所示：

```
public class BaseController {
  protected HttpServletRequest request;
  protected HttpServletResponse response;
  protected HttpSession session;
    @ModelAttribute
  public void setReqAndRes(HttpServletRequest request, HttpServletResponse response){
    this.request = request;
    this.response = response;
    this.session = request.getSession();
  }
}
```

我们可以把这个@ModelAttribute 特性，应用在 BaseController 当中，所有的 Controller 继承 BaseController，即在调用 Controller 时，先执行@ModelAttribute 方法。比如权限的验证(也可以使用 Interceptor)等。上述 HttpServletRequest，HttpServletResponse 的获取通过直接设置形参的方式，Spring 会自动将对应的对象传递给对应的形参，实际上 HttpSession 也可以通过设置形参自动传入的方式获取。

5.4.2 Controller 传递参数到前台

方式一：直接通过 request 对象传递。

前面我们讲到可以在 controller 中获取 request 对象，之后我们可以调用 setAttribute 方法将数据设置到 request 对象里，然后使用转发的方式进入 jsp 再通过调用 getAttribute 把值取出来即可。

方式二：直接通过返回值 ModelAndView 对象传递。

将方法的返回值放在 ModelAndView 里返回时，将数据存储在 ModelAndView 对象中，如：new ModelAndView("showUser", "message", message)，代码如下所示：

```
@RequestMapping("/showUser")
public ModelAndView showUser() {
    ModelAndView modelAndView = new ModelAndView("ShowUser", "message", "test message!");
    return modelAndView;
}
```

上面代码使用 ModelAndView 对象将数据传递到前台，ModelAndView 对象有三个参数，其中第一个参数为 url，第二个参数为要传递的数据的 key，第三个参数为数据对象，当要传递多个参数时可以多次调用 modelAndView.addObject("attributeName", attributeValue)，addObject 里传入的参数的值 attributeValue 是 Object 类型，所以理论上说可以传入任何类型的值。在这里要注意数据被默认是存放在 request 中的，在前台获取的数据的方式是：

```
${requestScope.message}
${requestScope.attributeName}
```

方式三：直接通过参数列表中添加形参 ModelMap 传递。

当向前台传递多个参数时，除了可以多次调用 modelAndView 的 addObject 外，我们也可以用 ModelMap 的方式。首先在方法的参数列表中添加形参 ModelMap，Spring 将自动创建 ModelMap 对象，然后调用 ModelMap 的 put(key，value)或者 addAttribute(key，value)将数据放入 ModelMap 中，Spring 自动将数据存入 request。代码如下所示：

```
@RequestMapping("/showUser")
public String showUser(ModelMap map) {
    map.put("message", "test message!");
    map.addAttribute("attributeName", "attributeValue");
    return "ShowUser";
}
```

在前台获取数据的方式和 ModelAndView 是一样的，这里就不再赘述。

在 SpringMVC 中，controller 中方法的返回值除了可以返回我们上面见过的 String、ModelAndView 和 ModelMap 三种类型外，还可以有 void、Map、Object、List、Set 等类型。返回值类型为 void 时则只是纯粹地执行了方法中的程序，程序响应的视图 URL 依然为请

求的视图 URL，返回值类型为 Map、Object、List、Set 等类型时，Spirng 会将返回的对象自动存储在 request 中。和 void 类型一样，程序响应的视图 URL 依然为请求的视图 URL。

5.5 SpringMVC 文件上传和下载

5.5.1 文件上传

SpringMVC 文件的上传需要借助 Apache 的 Commons 里两个工具类、配置文件上传处理 Bean、文件上传处理 Controller、文件上传的 Form 表单。下面分别介绍。

(1) 借助 Apache Commons 里两个工具类，则需导入 commons-fileupload-xxx.jar 和 commons-io-xxx.jar 两个包，下载地址都在 http://commons.apache.org/里可以找到。

(2) 配置文件上传处理 Bean 只需在 spring-mvc.xml 里配置 CommonsMultipartResolver 即可，配置代码如下所示：

```
<bean id = "multipartResolver"          class                                    =
"org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <property name = "defaultEncoding" value = "utf-8" />
        <property name = "maxInMemorySize" value = "10240" />
        <property name = "uploadTempDir" value = "/temp/" />
        <property name = "maxUploadSize" value = "-1" />
    </bean>
```

其中：

defaultEncoding：编码方式设置；

maxInMemorySize：最大内存大小；

uploadTempDir：临时文件存储目录；

maxUploadSize：上传最大文件大小，-1 为无限止。

(3) 文件上传处理 Controller。此 Controller 用于处理视图层文件流的获取并将获取的文件流写入本地文件中。代码如下所示：

```
@Controller
@RequestMapping("/file")
public class FileController {
    @RequestMapping("/initFileUpload")
    public String initFileUpload() {
        return "FileUpload";
    }
    @RequestMapping(value = "/upload", method = RequestMethod.POST)
    public String fileUpLoad(String name, @RequestParam("file") CommonsMultipartFile file, HttpSession
session) {
```

```
        if (!file.isEmpty()) {
            //获取上下文路径
            String path = session.getServletContext().getRealPath("/upload/");
            //获取文件名
            String fileName = file.getOriginalFilename();
            //获取文件类型
            String fileType = fileName.substring(fileName.lastIndexOf("."));
            //生成保存目标文件
            File targetFile = new File(path, new Date().getTime() + fileType);
            try {
                //向目标文件写入内容
                file.getFileItem().write(targetFile);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return "FileUpload";
    }
}
```

上面文件的取得是通过直接在处理的方法中设置形参@RequestParam("file") CommonsMultipartFile file，使用@RequestParam 对传入参数指定参数名，对于@RequestParam 可以通过 required = false 或者 true 来要求@RequestParam 配置的前端参数是否一定要传。如@RequestParam(value = " file", required = true)时如果不传参数会出错。上面我们实现的是单个文件的上传，如果要实现多文件的上传，可以将@RequestParam("file") CommonsMultipartFile file 参数修改成@RequestParam("file")CommonsMultipartFile[] files，接受多文件参数值即可。

(4) 文件上传的 Form 表单。代码如下：

```
<form action = "upload" method = "post" enctype = "multipart/form-data">
        <input type = "file" name = "file"><br> <input type = "submit"
        value = "submit">
    </form>
```

注意：在文件上传的 Form 表单里 method 值必须是 post 方式，enctype 值必须是 post 方式 multipart/form-data。

5.5.2 文件下载

文件下载可以使用最普通的方式实现，如要下载位于/upload 目录里的 fileName 文件，首先可获取要下载文件的输入流，并将获取的输入流读入缓冲流 BufferedInputStream，最后将缓冲流通过循环的方式写入到 response 的输出流实现文件下载功能，代码片段如下：

```
@RequestMapping(value = "/fileDownLoad", method = RequestMethod.GET)
    public void fileDownLoad(HttpSession session, HttpServletResponse response, String fileName, boolean
isOnline)
    throws Exception {
        //获取文件路径
        String path = session.getServletContext().getRealPath("/upload/") + "\\" + fileName;
        File file = new File(path);
        System.out.println(path);
        //判断文件是否存在
        if (!file.exists()) {
            response.sendError(404, "您要下载的文件没找到");
            return;
        }
        //获取下载文件输入流
        BufferedInputStream bufIn = new BufferedInputStream(new FileInputStream(file));
        byte[] buff = new byte[1024];
        int len = -1;
        response.reset();
        if (isOnline) {
            URL u = new URL("file:///" + path);
            response.setContentType(u.openConnection().getContentType());
            response.setHeader("Content-Disposition", "inline;filename = " + fileName);
        } else {
            //设置内容类型
            response.setContentType("application/x-msdownload");
            //设置下载文件名
            response.setHeader("Content-Disposition", "attachment; filename = " + fileName);
        }
        //获取输出流
        OutputStream out = response.getOutputStream();
        //循环输出下载文件流
        while ((len = bufIn.read(buff)) != -1) {
            out.write(buff, 0, len);
            out.flush();
        }
        //关闭输入输出流
        bufIn.close();
        out.close();
    }
```

5.6 SpringMVC 常用注解

SpringMVC 从 2.5 版本开始在编程中引入注解，除了前面我们使用过的@Controller、@RequestMapping、@RequestParam 和@ModelAttribute 注解，SpringMVC 还提供了大量不同功能的注解，SpringMVC 的注解随 Spring 版本的不断升级而不断扩展。下面是 SpringMVC 中的常用注解。

1. @Controller

Controller 控制器负责处理由 DispatcherServler 分发过来的请求，它把用户请求的数据经过业务处理层处理之后封装成一个 Model，然后再把该 Model 返回给对应的 View 进行展示。SpringMVC 使用@Controller 定义控制器，它还允许自动检测定义在类路径下的组件并自动注册。如想自动检测生效，需在 XML 头文件下引入 spring-context，配置代码如下所示：

```
<?xml version = "1.0" encoding = "UTF-8"?><beans xmlns = "http://www.springframework.org/schema/beans"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p = "http://www.springframework.org/schema/p"
  xmlns:context = "http://www.springframework.org/schema/context"
  xsi:schemaLocation = "
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">
 <context:component-scan base-package = "com.test.controller"/>
<!-- ... -->
</beans>
```

2. @RequestMapping

我们可以用@RequestMapping 注解将类似“/user”这样的 URL 映射到整个类或特定的处理方法上。一般来说，类级别的注解映射特定的请求路径到表单控制器上，而方法级别的注解只是映射为一个特定的 HTTP 方法请求(“GET”，“POST”等)或 HTTP 请求参数。代码如下：

```
@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping("/initAddUser")
    public String initAddUser() {
        return "AddUser";
    }
```

```
    @RequestMapping("/showUser")
    public String showUser(ModelMap map) {
        map.put("message", "test message!");
        map.addAttribute("attributeName", "attributeValue");
        return "ShowUser";
    }
    @RequestMapping(value = "/{userId}/deleteUser", method = RequestMethod.GET)
    public String deleteUser(@PathVariable String userId) {
        System.out.println("delete:" + userId);
        return "ShowUser";
    }
    @RequestMapping(value = "/{userBirth:\\d{4}-\\d{2}-\\d{2}}/updateUser")
    public String updateUser(@PathVariable String userBirth) {
        System.out.println("userBirth:" + userBirth);
        return "ShowUser";
    }
    @RequestMapping(value = "/addUser", method = RequestMethod.POST)
    public String addUser(Model model, User user, @ModelAttribute("name") String nickname) {
        System.out.println("name--:" + nickname + "\tage:" + user.getAge());
        return "ShowUser";
    }
}
```

@RequestMapping 既可以作用在类级别上，也可以作用在方法级别上。当它定义在类级别时，标明该控制器处理所有的请求都被映射到/user 路径下。@RequestMapping 中可以使用 method 属性标记其所接受的方法类型，如果不指定方法类型的话，可以使用 HTTP GET/POST 方法请求数据，但是一旦指定方法类型，就只能使用该类型获取数据。

@RequestMapping 可以使用 @Validated 与 BindingResult 联合验证输入的参数，在验证通过或失败的情况下，分别返回不同的视图。

@RequestMapping 支持使用 URI 模板访问 URL。URI 模板是像 URL 模样的字符串，由一个或多个变量名字组成，当这些变量有值的时候，它就变成了 URI。

3. @PathVariable

在 SpringMVC 中，可以使用@PathVariable 注解方法参数并将其绑定到 URI 模板变量的值上。URI 模板“/{userId}/deleteUser”指定变量的名字 userId，当控制器处理这个请求的时候，userId 的值会被设定到 URI 中。比如，当有一个像“123/ deleteUser”这样的请求时，userId 的值就是 123。

@PathVariable 可以有多个注解，代码如下：

```
@RequestMapping(value = "/{userId}/deleteUser/group/{groupId}", method = RequestMethod.GET)
    public String deleteUser(@PathVariable String userId, @PathVariable String groupId) {
```

```
    System.out.println("delete: userId" + userId+"groupId"+groupId);
    return "ShowUser";
}
```

@PathVariable 中的参数可以是任意的简单类型，如 in、long、Date 等。Spring 自动将其转换成合适的类型或者抛出 TypeMismatchException 异常。当然，我们也可以注册支持另外的数据类型。

如果@PathVariable 使用 Map<String, String>类型的参数时，Map 会填充到所有的 URI 模板变量中。

@PathVariable 支持使用正则表达式，因而功能强大，它能在路径模板中使用占位符，可以设定特定的前缀匹配、后缀匹配等自定义格式。

@PathVariable 还支持矩阵变量，因为现实中用得不多，就不详细介绍了。

4. @RequestParam

@RequestParam 将请求的参数绑定到方法中的参数上，如@RequestParam String inputStr。其实，即使不配置该参数，注解也会默认使用该参数。如果想定义指定参数必须传入，可以将@RequestParam 的 required 属性设置为 true(如@RequestParam(value = "id", required = true))。

5. @RequestBody

@RequestBody 是指方法参数应该被绑定到 HTTP 请求 Body 上。代码如下：

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
public void handle(@RequestBody String body, Writer writer) throws IOException {
    writer.write(body);
}
```

如果觉得@RequestBody 不如@RequestParams 使用方便，我们可以使用 HttpMessageConverter 将 request 的 body 转移到方法参数上，HttMessageConverser 可以将 HTTP 请求消息在 Object 对象之间互相转换，但一般情况下不会这么做。事实证明，@RequestBody 在构建 REST 架构时，比@RequestParam 有着更大的优势。

6. @ResponseBody

@ResponseBody 与@RequestBody 类似，它的作用是将返回类型直接输入到 HTTP response body 中。@ResponseBody 在输出 JSON 格式的数据时，会经常用到如下代码：

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
@ResponseBody public String helloWorld() {
return "Hello World";
}
```

7. @RestController

我们经常见到一些控制器实现了 REST 的 API，只为服务于 JSON，XML 或其他自定义的类型内容，@RestController 用来创建 REST 类型的控制器与@Controller 类型。

@RestController 就是这样一种类型，它避免了重复地写@RequestMapping 与@ResponseBody。代码如下：

```
@RestController
public class TestRestfulController {
    @RequestMapping(value = "/getUserName", method = RequestMethod.POST)
    public String getUserName(@RequestParam(value = "name") String name){
        return name;
    }
}
```

8. HttpEntity

HttpEntity 除了能获得 request 请求和 response 响应之外，它还能访问请求和响应头，代码如下所示：

```
@RequestMapping("/something")public ResponseEntity<String> handle(HttpEntity<byte[]> requestEntity)
throws UnsupportedEncodingException {
    String requestHeader = requestEntity.getHeaders().getFirst("MyRequestHeader"));
    byte[] requestBody = requestEntity.getBody();
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.set("MyResponseHeader", "MyValue");
    return new ResponseEntity<String>("Hello World", responseHeaders, HttpStatus.CREATED);
}
```

9. @ModelAttribute

@ModelAttribute 可以作用在方法或方法参数上，当它作用在方法上时，标明该方法的目的是添加一个或多个模型属性(model attributes)。该方法支持与@RequestMapping 一样的参数类型，但并不能直接映射成请求。控制器中的@ModelAttribute 方法会在调用@RequestMapping 方法之前被调用，代码如下所示：

```
@ModelAttribute
public Account addAccount(@RequestParam String number) {
    return accountManager.findAccount(number);
}
@ModelAttribute
public void populateModel(@RequestParam String number, Model model) {
    model.addAttribute(accountManager.findAccount(number));
    // add more ...
}
```

@ModelAttribute 方法用来在 model 中填充属性，如填充下拉列表、宠物类型或检索一个命令对象比如账户(用来在 HTML 表单上呈现数据)。

@ModelAttribute 方法有两种形式：一种是添加隐形属性并返回它；另一种是该方法接

受一个模型并添加任意数量的模型属性。用户可以根据自己的需要选择对应的形式。

当@ModelAttribute 作用在方法参数上时，表明该参数可以在方法模型中检索到。如果该参数不在当前模型中，则先将该参数实例化然后添加到模型中。一旦模型中有了该参数，则该参数的字段将填充所有请求参数匹配的名称中。这是 SpringMVC 中重要的数据绑定机制，它省去了单独解析每个表单字段的时间。

@ModelAttribute 是一种很常见的从数据库中检索属性的方法，它通过@Session Attributes 使用 request 请求存储。在一些情况下，可以很方便地通过 URI 模板的变量和类型转换器检索属性。

本章小结

本章我们主要学习了 SpringMVC，从 SpringMVC 概述开始到我们自己创建第一个 SpringMVC 程序和 SpringMVC RequestMapping 的基本设置、SpringMVC 前后台数据交互、SpringMVC 文件上传下载以及最后 SpringMVC 常用注解，对 SpringMVC 的学习使我们了解到 SpringMVC 是非常优秀的 MVC 框架，它不仅结构简单，而且功能强大而灵活，性能也很优越，我们利用 SpringMVC 可以很容易地写出性能优秀的程序。

练习题

一、选择题

1. 当一个 Web 请求发送到 SpringMVC 应用程序时，(　　)首先接收到该请求。

A. Servlet　　B. SpringMVC

C. DispatcherServlet　　D. Contorller

2. 在 SpringMVC 中定义一个控制器类，这个类必须标有(　　)注解。

A. @Controller　　B. @ResponseBody

C. @RequestMapping　　D. @SessionAttributes

3. 下列关于@RequestMapping 的说明(　　)是不对的。

A. 不能用来注解类

B. 可以注解控制器类的方法

C. 拥有 value 属性，该属性对应于 Web 请求地址

D. 拥有 method 属性，可以指定接受 Web 请求方法类型

4. 要使用 SpringMVC 的标签需要用(　　)引入对应的标签库。

A. <%@ taglib %>　　B. <%@ page %>

C. <%@ import %>　　D. <%@ link %>

5. 下列关于 SpringMVC 国际化的说法中(　　)是不正确的。

A. 需要创建.properties 的资源文件

B. 资源文件中的格式是“键值对”格式，即“键 = 值”

C．在 JSP 页面，不需要引入额外的标签库

D．静态文字的地方需要<spring:message>标签

6．在 SSM 框架中，SpringMVC 承担的责任是(　　)。

A．定义实体类　　B．数据的增删改查操作

C．业务逻辑的描述　　D．页面展示和控制转发

7．要启动 SpringMVC 框架，在 web.xml 文件中需要配置(　　)。

A．监听器　　B．SpringMVC

C．Controller　　D．DispatcherServlet

8．返回一个 JSON 对象，这个类必须标有(　　)注解。

A．@Controller　　B．@ResponseBody

C．@RequestMapping　　D．@SessionAttributes

9．关于以下两种@RequestMapping 的说明正确的选择是(　　)。

(1) 不能用来注解类

(2) 可以注解控制器类的方法

A．都正确　　B．只有 1 正确　　C．只有 2 正确　　D．都不正确

10．要使用<form:form>标签需要用(　　)引入标签库。

A．<%@ import %>　　B．<%@ page %>

C．<%@ taglib %>　　D．<%@ include%>

11．下列关于 SpringMVC 国际化的资源文件的格式说法中(　　)是正确的。

A．可以是 HTML 文件　　B．可以是 xml 文件

C．可以是 txt 文件　　D．可以是 properties 文件

二、填空题

1．当一个 Web 请求发送到 SpringMVC 应用程序，_____首先接收请求。然后组织那些在 Spring Web 应用程序中上下文配置的(例如实际请求处理控制器和视图解析器)或者使用注解配置的组件，所有的这些都需要处理该请求。

2．在 Spring 中定义一个控制器类，这个类必须标有_____注解。当有其注解的控制器收到一个请求时，它会寻找一个合适的_____方法去处理这个请求。为了这样做，一个控制器类的方法需要被_____注解装饰。

3．SpringMVC 在 Controller 控制层方法中通常返回的是_____，如何定位到真正的页面，就需要通过视图解析器，SpringMVC 里提供了多个视图解析器。

4．InternalResourceViewResolver 是比较常用的视图解析器，InternalResourceViewResolver 解析器会自动添加_____和_____，在上面的配置中，当处理器 Controller 中的方式返回逻辑视图字符串“index”时，它实际要定向的页面应该是_____。

5．SpringMVC 常常还通过_____或_____方式返回逻辑视图。

6. 在 SpringMVC 中自定义的 controller 会调用有_____注解字样的方法来处理请求。

7. 类上注解_____表示该类是一个处理请求和应答的 Controller，Controller 可以实现前后台数据交互。

8. 注解_____将请求的参数绑定到方法中的参数上。

9. 在 SpringMVC 中，可以使用 _____注解方法参数并将其绑定到 URI 模板变量的值上。

三、问答题

1. 列举 SpringMVC 的常用注解，并说明其作用。

2. SpringMVC 的运行流程是什么？

3. 简述使用 SpringMVC 框架进行编程的步骤。

第六章　SpringMVC Spring MyBatis 集成

前面几章中学习了 Spring、SpringMVC 和 MyBatis。我们知道，Spring 是一个轻量级的控制反转(IoC)和面向切面(AOP)的容器框架，它是为了解决企业应用开发的复杂性而创建的。Spring 使用基本的 JavaBean 来完成以前只能由 EJB 完成的事情，使企业应用开发变得简单高效，且可维护性得到极大提高。SpringMVC 是一个 MVC 的流程框架，SpringMVC 分离了控制器、模型对象、分派器以及处理程序对象的角色，这种分离让它们更容易进行定制，在流程处理方面更加灵活，可以很容易地进行扩展，可以和 Spring 框架进行无缝集成。MyBatis 是一个基于 Java 的持久层框架。MyBatis 提供的持久层框架包括 sql Maps 和 Data Access Objects(DAO)，MyBatis 消除了几乎所有的 JDBC 代码和参数的手工设置以及结果集的检索。MyBatis 使用简单的 XML 或注解用于配置和原始映射，将接口和 Java 的 POJOs(Plain Old Java Objects，普通的 Java 对象)映射成数据库中的记录。下面将对这三个框架(简称：SSM)的集成进行讲解。

本章知识要点

- 依赖包的引入；
- Spring 与 MyBatis 的集成；
- 集成 SpringMVC。

6.1　依赖包的引入

1. Spring 和 SpringMVC 包下载

进入 http://repo.spring.io/milestone/org/springframework/spring 选择要下载的 Spring 版本，最新的 Spring 已经更新到了 5 系列，为了和 SpringMVC 及 MyBatis 版本匹配使用，本章中使用的 Spring 的版本是 4.3.0，所下载的包为 spring-framework-4.3.0.RC2-dist.zip。

2. MyBatis 下载

进入 https://github.com/mybatis 或 http://code.google.com/p/mybatis(可能要通过代理方式)下载 MyBatis 的发布包，本章中使用的 MyBatis 的版本是 3.4.2，所下载的包为 mybatis-3.4.2.zip。

3. Spring 和 MyBatis 集成中间包下载

进入 http://www.mybatis.org/spring/ 下载集成中间包，本章使用的版本为 1.3.1，所下载的包为 mybatis-spring-1.3.1.zip。

4. 数据库驱动包下载

本章采用的是 MySQL5 作为数据库，在集成之前已经默认了该数据库。可以进入 http://www.mysql.com/products/connector/下载 mysql 的 java 驱动包 mysql-connector- java-5.1.40.zip。

5. 数据库连接池包及其依赖包下载

本章数据库连接池采用 Apache 的 DBCP2 方式实现，下载地址为 http://commons.apache.org/ proper/commons-dbcp/download_dbcp.cgi。本章使用的版本为 2.1.1，所下载的包为 commons-dbcp2-2.1.1-bin.zip.对于 DBCP2 它还依赖于 Apache 的 Commons Pool 公共开源包，下载地址为 http://commons.apache.org/proper/commons-pool/download_pool.cgi，下载包为 commons-pool2-2.4.2-bin.zip。

6. AspectJ 面向切面的框架

进入 http://www.eclipse.org/aspectj/downloads.php 框架包，本章使用的版本为 1.8.9，所下载的包为 aspectj-1.8.9.jar。

7. 日志包及日志依赖包下载

本章采用的是 Log4J 作为日志系统，可以进入 http://logging.apache.org/log4j/1.2/下载，本章使用的 Log4J 版本是 1.2.17，所下载的包为 log4J-1.2.17.zip. 对于 Log4J 它要完成日志输出还依赖于 Apache 的 Commons Logging 公共开源包，下载地址为 http://commons.apache.org/proper/commons-logging/，下载包为 commons-logging-1.2-bin.zip。

8. JSTL 可选标签库下载

为了便于读取服务端的值，我们这里使用 JSTL1.2，下载地址为 http://tomcat.apache.org/taglibs/standard/，下载包是 jstl-1.2.jar。

如果项目正在使用 maven，那么配置这些 jar 包依赖就变得简单多了。在 pom.xml 中添加以下依赖即可，代码如下：

```
<properties>
        <!-- spring版本号 -->
        <spring.version>4.3.0.RC2</spring.version>
        <!-- mybatis版本号 -->
        <mybatis.version>3.4.2</mybatis.version>
        <!-- log4j日志文件管理包版本 -->
        <slf4j.version>1.7.7</slf4j.version>
        <log4j.version>1.2.17</log4j.version>
    </properties>
    <dependencies>
<!-- junit测试包 -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
```

```
            <version>4.11</version>
            <!-- 表示开发的时候引入，发布的时候不会加载此包 -->
            <scope>test</scope>
        </dependency>
        <!-- spring核心包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
        </dependency>
<!-- spring的web支持包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.version}</version>
        </dependency>
<!-- spring O/X 映射支持包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-oxm</artifactId>
            <version>${spring.version}</version>
        </dependency>
<!-- spring注解事务映射支持包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>${spring.version}</version>
        </dependency>
<!-- spring jdbc数据库访问支持包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <!-- SpringMVC支持包 -->
<dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>${spring.version}</version>
```

```
        </dependency>
<!-- spring 面向切面编程支持包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>${spring.version}</version>
        </dependency>
<!-- spring核心扩展支持包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
            <version>${spring.version}</version>
        </dependency>
<!-- spring测试相关支持包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <!-- mybatis核心包 -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>${mybatis.version}</version>
        </dependency>
        <!-- mybatis/spring集成支持包 -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>1.3.1</version>
        </dependency>
        <!-- 导入java ee jar 包 -->
        <dependency>
            <groupId>javax</groupId>
            <artifactId>Java EE-api</artifactId>
            <version>8.0</version>
        </dependency>
        <!-- 导入Mysql数据库驱动包 -->
        <dependency>
```

```
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.40</version>
        </dependency>
        <!-- 导入dbcp的jar包，用来在applicationContext.xml中配置数据库 -->
        <dependency>
            <groupId>commons-dbcp2</groupId>
            <artifactId>commons-dbcp2</artifactId>
            <version>2.1.1</version>
        </dependency>
        <!-- JSTL标签类 -->
        <dependency>
            <groupId>jstl</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
        </dependency>
        <!-- 日志文件管理包 -->
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>${log4j.version}</version>
        </dependency>
        <!-- 映入JSON -->
        <dependency>
            <groupId>org.codehaus.jackson</groupId>
            <artifactId>jackson-mapper-asl</artifactId>
            <version>1.9.13</version>
        </dependency>
        <!-- 上传组件包 -->
        <dependency>
            <groupId>commons-fileupload</groupId>
            <artifactId>commons-fileupload</artifactId>
            <version>1.3.1</version>
        </dependency>
<!-- IO流工具类包 -->
        <dependency>
            <groupId>commons-io</groupId>
            <artifactId>commons-io</artifactId>
            <version>2.4</version>
```

```
        </dependency>
<!-- 处理常用的编码工具类包 -->
        <dependency>
            <groupId>commons-codec</groupId>
            <artifactId>commons-codec</artifactId>
            <version>1.9</version>
        </dependency>
    </dependencies>
```

本章中采用直接复制 jar 的方式。把上面所下载的压缩包解开，把相应的 jar 复制到 lib 下，如图 6-1 所示。

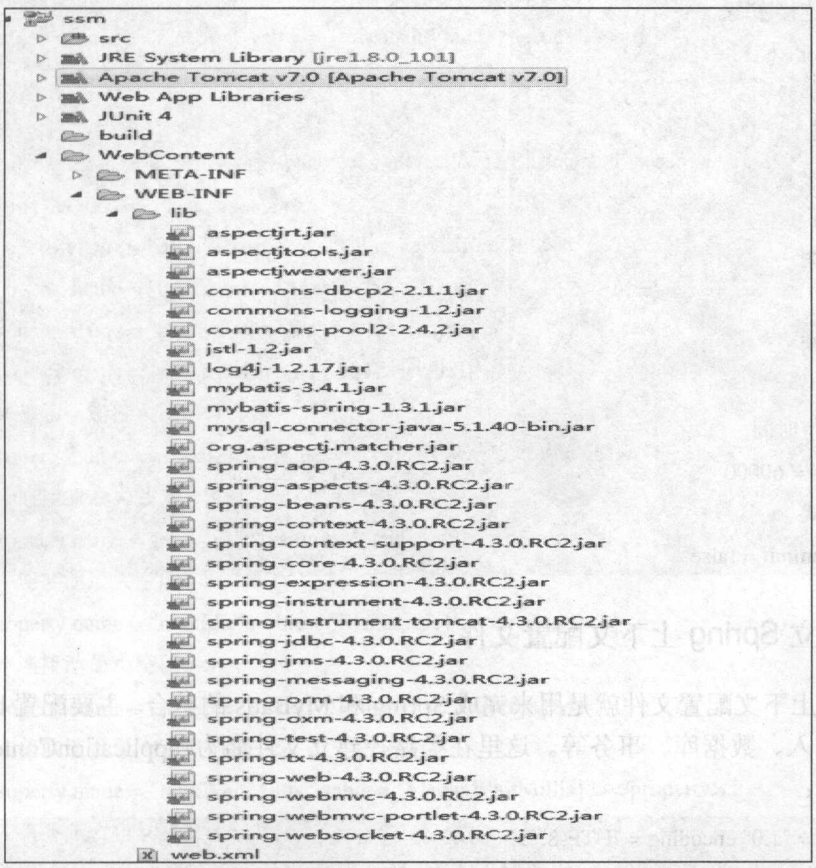

图 6-1　lib 下 jar 包

6.2　Spring 与 MyBatis 的集成

所有需要的 jar 包都引入以后，接下来进行 Spring 与 MyBatis 的整合，然后用 JUnit 进行测试集成的结果。集成步骤如下。

6.2.1 建立 JDBC 属性文件

JDBC 属性文件是关于数据库相关的配置信息，这里在类路径建立文件名为 jdbc.properties，以下是详细配置信息：

```
#定义数据库连接驱动类
driver = com.mysql.jdbc.Driver
#定义数据库连接URL
url = jdbc:mysql:///mydb
#定义数据库连接用户名
username = root
#定义数据库连接密码
password = root
#定义初始连接数
initialSize = 0
#定义最大连接数
maxTotal = 20
#定义最大空闲
maxIdle = 20
#定义最小空闲
minIdle = 1
#定义最长等待时间
maxWaitMillis = 60000
#默认提交方式
defaultAutoCommit = false
```

6.2.2 建立 Spring 上下文配置文件

Spring 上下文配置文件就是用来完成 Spring 和 MyBatis 的整合。主要配置 bean 自动扫描、依赖注入、数据库、事务等。这里在类路径建立文件名为 applicationContext.xml，代码如下所示：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop = "http://www.springframework.org/schema/aop"
    xmlns:context = "http://www.springframework.org/schema/context"
    xmlns:tx = "http://www.springframework.org/schema/tx"
    xsi:schemaLocation = "http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
```

```
                                http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
                                http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
                                http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd">
    <!-- 自动扫描包 -->
    <context:component-scan base-package = "com.test" />
    <!-- 引入上一节配置jdbc.properties数据库信息文件 -->
    <bean id = "propertyConfigurer"          class                                    =
"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name = "location" value = "classpath:jdbc.properties" />
    </bean>
<!-- dbcp数据源配置 -->
    <bean id = "dataSource" class = "org.apache.commons.dbcp2.BasicDataSource"
        destroy-method = "close">
        <property name = "driverClassName" value = "${driver}" />
        <property name = "url" value = "${url}" />
        <property name = "username" value = "${username}" />
        <property name = "password" value = "${password}" />
        <!-- 初始化连接大小 -->
        <property name = "initialSize" value = "${initialSize}"></property>
        <!-- 连接池最大数量 -->
        <property name = "maxTotal" value = "${maxTotal}"></property>
        <!-- 连接池最大空闲 -->
        <property name = "maxIdle" value = "${maxIdle}"></property>
        <!-- 连接池最小空闲 -->
        <property name = "minIdle" value = "${minIdle}"></property>
        <!-- 获取连接最大等待时间 -->
        <property name = "maxWaitMillis" value = "${maxWaitMillis}"></property>
        <!-- 事务是否自动提交 -->
        <property name = "defaultAutoCommit" value = "${defaultAutoCommit}"></property>

    </bean>
    <!-- spring和MyBatis完美整合，不需要mybatis的配置映射文件 -->
    <bean id = "sqlSessionFactory" class = "org.mybatis.spring.SqlSessionFactoryBean">
        <property name = "dataSource" ref = "dataSource" />
        <!-- 自动扫描mapping.xml文件 -->
        <property name = "mapperLocations" value = "classpath:com/test/mapper/*.xml"></property>
```

```
    </bean>

    <!--Mapper接口调用方式要用，如果不用mapper接口方式采用实体dao，可以不配接口所在包名，
Spring会自动查找其下的类 -->
    <bean class = "org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name = "basePackage" value = "com.test.mapper" />
        <property name = "sqlSessionFactoryBeanName" value = "sqlSessionFactory"></property>
    </bean>
    <!--采用实体 dao 调用方式要用，在 dao 里可以用注解@Resource(name = "sqlSessionTemplate")
方式得到或用 ao 继承 SqlSessionDaoSupport 再采用 setter 方式对 dao 进行注入-->
    <bean id = "sqlSessionTemplate" class = "org.mybatis.spring.SqlSessionTemplate">
        <constructor-arg ref = "sqlSessionFactory" />
    </bean>
    <!-- (事务管理器)transaction manager, use JtaTransactionManager for global tx -->
    <bean id = "transactionManager"
        class = "org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name = "dataSource" ref = "dataSource" />
    </bean>
    <!-- (AOP事务管理方法匹配规则) -->
    <tx:advice id = "txAdvice" transaction-manager = "transactionManager">
        <tx:attributes>
            <tx:method name = "delete*" propagation = "REQUIRED" read-only = "false"
                rollback-for = "java.lang.Exception"/>
            <tx:method name = "insert*" propagation = "REQUIRED" read-only = "false"
                rollback-for = "java.lang.Exception" />
            <tx:method name = "update*" propagation = "REQUIRED" read-only = "false"
                rollback-for = "java.lang.Exception" />
            <tx:method name = "save*" propagation = "REQUIRED" read-only = "false"
                rollback-for = "java.lang.Exception" />
            <tx:method name = "*" propagation = "SUPPORTS" read-only = "true"/>

        </tx:attributes>
    </tx:advice>
    <!-- AOP事务处理 -->
    <aop:config>
        <aop:pointcut id = "allServiceMethods" expression = "execution(* com.test.service..*(..))" />
        <aop:advisor pointcut-ref = "allServiceMethods" advice-ref = "txAdvice" />
    </aop:config>
</beans>
```

6.2.3　Log4J 的配置

为了方便调试，一般都会使用日志来输出信息，Log4J 是 Apache 的一个开放源代码项目，通过使用 Log4J，可以控制日志信息输送的目的地为控制台、文件、GUI 组件，甚至是套接口服务器、NT 的事件记录器、UNIX Syslog 守护进程等；我们还可以控制每一条日志的输出格式；通过定义每一条日志信息的级别，还能够更加细致地控制日志的生成过程。下面是基本的配置：

```
#定义LOG输出级别
log4j.rootLogger = INFO, Console, File
#定义日志输出目的地为控制台
log4j.appender.Console = org.apache.log4j.ConsoleAppender
log4j.appender.Console.Target = System.out
#可以灵活地指定日志输出格式，下面一行是指定具体的格式
log4j.appender.Console.layout = org.apache.log4j.PatternLayout
log4j.appender.Console.layout.ConversionPattern = [%c] - %m%n
#文件大小到达指定尺寸的时候产生一个新的文件
log4j.appender.File = org.apache.log4j.RollingFileAppender
#指定输出目录
log4j.appender.File.File = /logs/ssm.log
#定义文件最大大小
log4j.appender.File.MaxFileSize = 10MB
# 输出所有日志，如果换成DEBUG表示输出DEBUG以上级别日志
log4j.appender.File.Threshold = ALL
log4j.appender.File.layout = org.apache.log4j.PatternLayout
log4j.appender.File.layout.ConversionPattern = [%p] [%d{yyyy-MM-dd HH\:mm\:ss}][%c]%m%n
```

6.2.4　JUnit 测试

经过上面的步骤已经完成了 Spring 和 MyBatis 的基本整合，接下来将编写测试代码来测试环境是否已经成功。具体步骤如下：

(1) 创建测试数据表，插入样本数据，代码如下所示：

```
DROP TABLE IF EXISTS 'user';
CREATE TABLE 'user' (
  'id' int(11) NOT NULL AUTO_INCREMENT,
  'user_name' varchar(40) NOT NULL,
  'password' varchar(255) NOT NULL,
  'age' int(4) NOT NULL,
  PRIMARY KEY ('id')
```

```
) ENGINE = InnoDB AUTO_INCREMENT = 6 DEFAULT CHARSET = utf8;
INSERT INTO 'user' VALUES ('1', '张三', 'test', '22');
INSERT INTO 'user' VALUES ('2', '李四', 'aa', '20');
INSERT INTO 'user' VALUES ('3', '王五', 'aa', '20');
```

(2) 建立域模型文件代码。

域模型用于表示对象和它拥有的属性和操作数据，在 com.test.domain 包下建立 User.java，代码如下所示：

```
package com.test.domain;
public class User {
    private Integer id;
    private String userName;
    private String password;
    private Integer age;
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName == null ? null : userName.trim();
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password == null ? null : password.trim();
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
}
```

(3) 建立映射文件。

映射文件用来处理 User 对象属性和 User 表字段之间的数据映射对应方式，在 com.test.mapper 包下建立 UserMapper.xml 映射文件，代码如下所示：

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
                                "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace = "com.test.mapper.UserMapper">
<!-- User对象User表之间的映射方式 -->
    <resultMap id = "BaseResultMap" type = "com.test.domain.User">
        <id column = "id" property = "id" jdbcType = "INTEGER" />
    <result column = "user_name" property = "userName" jdbcType = "VARCHAR" />
    <result column = "password" property = "password" jdbcType = "VARCHAR" />
        <result column = "age" property = "age" jdbcType = "INTEGER" />
    </resultMap>
<!-- 以ID为条件进行查询User表 -->
    <select id = "getUserById" resultMap = "BaseResultMap"
        parameterType = "java.lang.Integer">
        select
        id, user_name, password, age
        from user
        where id = #{id}
    </select>
</mapper>
```

(4) 建立持久层 Dao 文件。

Dao 层用于处理应用和数据库间进行数据的交互，在包 com.test.dao 下建立 UserDao.java，代码如下所示：

```
package com.test.dao;
import javax.annotation.Resource;
import org.mybatis.spring.SqlSessionTemplate;
import org.springframework.stereotype.Repository;
import com.test.domain.User;
@Repository("userDao")
public class UserDao{
    @Resource(name = "sqlSessionTemplate")
    private SqlSessionTemplate sqlSessionTemplate;
    public User getUserById(int id){
        return (User) sqlSessionTemplate.selectOne("com.test.mapper.UserMapper.getUserById", id);
    }
}
```

(5) 建立服务层 Service 接口和实现。

服务层用于处理 Controller 层和 Dao 层之间的数据交互，这里采用接口的方式进行编码，在包 com.test.service 下建立接口类 UserService.java，代码如下所示：

```
package com.test.service;
import com.test.domain.User;
public interface UserService {
    public User getUserById(int id);
}
```

在包 com.test.service 下建立接口实现类 UserServiceImpl.java，代码如下所示：

```
package com.test.service;
import javax.annotation.Resource;
import org.springframework.stereotype.Service;
import com.test.dao.UserDao;
import com.test.domain.User;
import com.test.mapper.UserMapper;
@Service("userService")
public class UserServiceImpl implements UserService {
    @Resource
    private UserDao userDao;
    @Override
    public User getUserById(int id) {
        return this.userDao.getUserById(id);
    }
}
```

(6) 建立测试类。

建立测试类 TestCase.java，代码如下所示：

```
package com.test.app;
import javax.annotation.Resource;
import org.apache.log4j.Logger;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import com.test.domain.User;
import com.test.service.UserService;
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:applicationContext.xml" })
public class TestCase {
```

```
    private static Logger logger = Logger.getLogger(TestCase.class);
    @Resource
    private UserService userService = null;
    @Test
    public void test1() {
        System.out.println(userService);
        User user = userService.getUserById(1);
        logger.info(user.getUserName());
    }
}
```

运行测试类，如果显示如图 6-2 所示说明我们的 Spring 和 MyBatis 已经正常集成。

```
[org.springframework.context.support.GenericApplicationContext] - Refreshing org.springframework.context.support.GenericApplicationContext
[org.springframework.beans.factory.config.PropertyPlaceholderConfigurer] - Loading properties file from class path resource [jdbc.properti
com.test.service.UserServiceImpl@4416d64f
org.apache.ibatis.binding.MapperProxy@14f9390f
----------------------------------
[com.test.app.TestCase] - 张三
[org.springframework.context.support.GenericApplicationContext] - Closing org.springframework.context.support.GenericApplicationContext@594
```

图 6-2　测试结果

6.3　集成 SpringMVC

上面已经完成了 Spring 和 MyBatis 的集成，接下来将把 SpringMVC 继续集成进来，要集成 SpringMVC，首先要添加关于 SpringMVC 的配置文件 spring-mvc.xml 和配置 web.xml 文件全局加载项。下面我们按步骤进行。

6.3.1　建立 SpringMVC 配置文件

spring-mvc.xml 配置文件主要用于配置 SpringMVC 视图解析器和 Controller 的加载方式及部分静态资源等，代码如下所示：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance" xmlns:p = "http://www.springframework.org/
                schema/p"
    xmlns:context = "http://www.springframework.org/schema/context"
    xmlns:mvc = "http://www.springframework.org/schema/mvc"
    xsi:schemaLocation = "http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
                http://www.springframework.org/schema/context
                http://www.springframework.org/schema/context/spring-context-3.1.xsd
```

```
                    http://www.springframework.org/schema/mvc
                    http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd">
    <!-- 自动扫描该包，使SpringMVC认为包下用了@controller注解的类是控制器 -->
    <context:component-scan base-package = "com.test.controller" />
<!-- if you use annotation you must configure following setting -->
    <mvc:annotation-driven />
      <!--
       配置静态资源，直接映射到对应的文件夹，不被DispatcherServlet处理
    -->
    <!-- <mvc:resources mapping = "/img/**" location = "/img/" />
    <mvc:resources mapping = "/js/**" location = "/js/" />
    <mvc:resources mapping = "/css/**" location = "/css/" />
    <mvc:resources mapping = "/html/**" location = "/html/" /> -->
      <!-- 定义跳转的文件的前后缀 ，视图模式配置-->
      <bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
          <!-- 这里的配置是自动给后面action的方法return的字符串加上前缀和后缀，变成一个可用的
url地址 -->
          <property name = "prefix" value = "/Web-INF/jsp/" />
          <property name = "suffix" value = ".jsp" />
      </bean>
</beans>
```

6.3.2 配置 web.xml 文件

在 web.xml 里主要配置字符串编码处理过滤器、Spring 和 SpringMVC 的加载方式。代码如下：

```
<?xml version = "1.0" encoding = "UTF-8"?>
<web-app xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance" xmlns =
"http://java.sun.com/xml/ns/Java EE"
xsi:schemaLocation = "http://java.sun.com/xml/ns/Java EE http://java.sun.com/xml/ns/Java EE/web-app_3_0.xsd"
version = "3.0">
  <display-name>Archetype Created Web Application</display-name>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
  </context-param>
  <filter>
    <filter-name>encodingFilter</filter-name>
<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
```

```
    <async-supported>true</async-supported>
    <init-param>
      <param-name>encoding</param-name>
      <param-value>UTF-8</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>encodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <listener>    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <listener>    <listener-class>org.springframework.web.util.IntrospectorCleanupListener</listener-class>
  </listener>
  <servlet>
    <servlet-name>SpringMVC</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:spring-mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
    <async-supported>true</async-supported>
  </servlet>
  <servlet-mapping>
    <servlet-name>SpringMVC</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>/index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

关于上面配置：

(1) encodingFilter 过滤器，主要是用来处理字符串编码，在 Spring 框架中从页面传来的字符串的编码问题主要交给过滤器 CharacterEncodingFilter 进行处理，这个过滤器就是针对每次浏览器请求进行过滤，然后在其之上添加了父类没有的功能即处理字符编码。其中 encoding 用来设置编码格式，forceEncoding 用来设置是否理会 request.getCharacterEncoding() 方法，设置为 true 则强制覆盖之前的编码格式。

(2) 在 web.xml 里配置的监听器 ContextLoaderListener，其作用就是启动 Web 容器时，自动装配 ApplicationContext 的配置信息。因为它实现了 ServletContextListener 这个接口，在 web.xml 配置这个监听器，启动容器时，就会默认执行它实现的方法。在 ContextLoaderListener 中关联了 ContextLoader 这个类，所以整个加载配置过程由 ContextLoader 来完成。

(3) <param-name>contextConfigLocation</param-name>配置是指明了 Spring 加载上下文是引用的 xml 文件，如果在 Web.xml 中不写任何参数配置信息，默认的路径是"/WEB-INF/applicationContext.xml，在 WEB-INF 目录下创建的 xml 文件的名称必须是 applicationContext.xml。如果要自定义文件名可以在 web.xml 里加入 contextConfigLocation 这个 context 参数，在<param-value> </param-value>里指定相应的 xml 文件名，如果有多个 xml 文件，可以写在一起并以“,”号分隔。也可以这样 applicationContext-*.xml 采用通配符，比如目录下有 applicationContext-mybatis-base.xml、applicationContext-action.xml、applicationContext-mybatis-dao.xml 等文件，都会一同被载入。在 ContextLoaderListener 中关联了 ContextLoader 这个类，所以整个加载配置过程由 ContextLoader 来完成。

(4) DispatcherServlet。要使用 SpringMVC，配置 DispatcherServlet 是第一步。DispatcherServlet 是一个 Servlet，所以可以配置多个 DispatcherServlet。DispatcherServlet 是前置控制器，配置在 web.xml 文件中。目的是拦截匹配的请求，servlet 拦截匹配的规则要自己定义，把拦截下来的请求，依据某规则分发到目标 Controller(我们写的 Controller)来处理。在 DispatcherServlet 的初始化过程中，框架会在 Web 应用的 WEB-INF 文件夹下寻找名为[servlet-name]-servlet.xml 的配置文件，生成文件中定义的 Bean，同时也指明了配置文件的文件名，不使用默认配置文件名，而使用 spring-mvc.xml 配置文件。

<load-on-startup>1</load-on-startup>是启动顺序，让这个 Servlet 随 Servletp 容器一起启动。

(5) <url-pattern>/</url-pattern>表示 SpringMVC 会拦截 URL 中带“/”的请求。Servlet 拦截匹配规则可以自己定义，以映射为@RequestMapping("/user/add")时为例，拦截哪种 URL 合适？第一种方案为拦截*.do、*.htm，例如：/user/add.do，这是最传统的方式，最简单也最实用，不会导致静态文件(jpg, js, css 等)被拦截。第二种方案拦截/，例如：/user/add，可以实现现在很流行的 REST 形式。很多互联网类型的应用选择这种形式的 URL。弊端是会导致静态的文件(jpg, js, css)被拦截后不能正常显示，这时可以通过在 spring-mvc.xml 里配置静态资源的方式，也可以激活 Tomcat 的 defaultServlet 来处理静态文件，但要写在 DispatcherServlet 配置的前面，让 defaultServlet 先拦截请求，这样请求就不会进入 Spring 了，如下为两种处理方式代码。

spring-mvc.xml 配置静态资源：

```
<mvc:resources mapping = "/img/**" location = "/img/" />
    <mvc:resources mapping = "/js/**" location = "/js/" />
    <mvc:resources mapping = "/css/**" location = "/css/" />
    <mvc:resources mapping = "/html/**" location = "/html/" />
```

激活 Tomcat 的 defaultServlet 来处理静态资源：

```
<servlet-mapping>
```

```
        <servlet-name>default</servlet-name>
        <url-pattern>*.css</url-pattern>
</servlet-mapping>
```

6.3.3　测试

经过上面的步骤我们基本完成了 Spring、SpringMVC 和 MyBatis 的整合，接下来我们将编写测试代码来测试是否已经成功。

(1) 新建控制层 Controller 类。

控制层负责处理视图层和模型层之间的数据交互，在 com.test.controller 包下新建控制层 UserController.java 类文件，代码如下所示：

```
package com.test.controller;
import javax.annotation.Resource;
import javax.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import com.test.domain.User;
import com.test.service.UserService;
@Controller
@RequestMapping("/user")
public class UserController {
    @Resource
    private UserService userService;
    @RequestMapping("/ShowUser")
    public String toIndex(HttpServletRequest request, Model model){
        int userId = Integer.parseInt(request.getParameter("id"));
        User user = this.userService.getUserById(userId);
        model.addAttribute("user", user);
        return "ShowUser";
    }
}
```

(2) 建立测试 JSP。

JSP 用于显示从控制层传回的数据，在 Web-INF/jsp 下新建 ShowUser.jsp 文件，用于显示查询的用户信息，代码如下所示：

```
<%@ page language = "java" import = "java.util.*" pageEncoding = "utf-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@ taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<html>
```

```
<head>
<title>测试</title>
</head>
<body>
    id:${user.id}<br>
    username:${user.userName}
</body>
</html>
```

(3) 部署运行。

部署运行后在浏览器输入地址 http://localhost:8080/ssm/user/ShowUser?id = 1，当出现如下界面时集成成功。如图 6-3 所示。

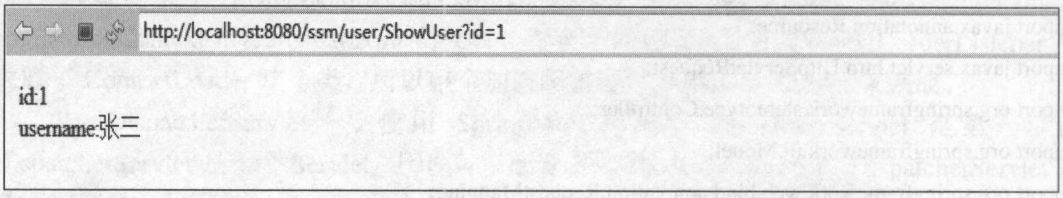

图 6-3　运行结果

至此我们对三个框架的整合就完成了。

本章小结

在本章中我们学习了怎样将 Spring、SpringMVC 和 MyBatis 与框架集成。Ssm 架构是标准的 MVC 模式，将整个系统划分为 View 表现层、Controller 层、Service 层、DAO 层，使用 SpringMVC 负责请求的转发和视图管理，Spring 实现业务对象管理，MyBatis 作为数据对象的持久化引擎。

练习题

一、问答题

1. 简述 Spring、SpringMVC、MyBatis 集成的意义和步骤。
2. 简述 SSM 集成后每个框架的职责和用途。

第七章 项目实战

7.1 项目的需求分析

云服务器是一种简单高效、安全可靠、处理能力可弹性伸缩的计算服务。服务器是云计算服务的重要组成部分，是面向各类互联网用户提供综合业务的服务平台。平台整合了传统意义上的互联网应用三大核心要素：计算、存储、网络，面向用户提供公用化的互联网基础设施服务。

云服务器服务包括两个核心产品：面向中小企业用户与高端用户的云服务器租用服务和面向大中型互联网用户的弹性计算平台服务。

随着信息化意识的不断加强，越来越多的企业开始着手建立自己的企业网站，并开展网络营销工作，作为国内企业的主体，数量众多的中小企业代表着庞大的互联网应用群体。以BAT为代表的互联网企业纷纷推出的云服务器租用服务很好地满足了各类中小企业用户的互联网需求。基于云服务器租用服务的形式多样，收费方式也是千变万化，云服务器租赁资费后台管理系统，英文全称 Cloud-Server Rent Cost Backend Management System，简称“CSRCBMS”，对租用服务器的资费进行管理。该系统具有的功能有：角色管理、管理员管理、资费管理、账务账号管理、业务账号管理、账单管理、报表、个人信息和修改密码。每个管理员登录成功后，都可以使用“个人信息”和“修改密码”功能，以实现个人信息和密码的维护，是否可以使用其他功能则取决于该管理员所拥有的权限。管理员登录成功后可以进行的操作完全取决于其所拥有的权限。CSRCBMS 系统会内置好一个“超级管理员”，作为使用该系统的第一位管理员用户。超级管理员成功登录系统后，则可以通过“角色管理”和“管理员管理”模块来创建其他管理员，以共同使用 CSRCBMS 系统。下面对系统的主要功能进行描述。

7.1.1 基础信息模块

1. 用例图

基础信息模块用例如图 7-1 所示。

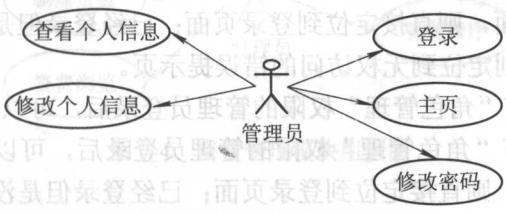

图 7-1 基础信息模块用例图

2. 用例描述

(1) 登录。管理员使用用户名和密码登录系统，如果登录用例成功，则进入主页。登录用例失败给出错误提示。

(2) 主页。主页是系统的功能导航页，显示当前操作员可以操作的功能链接，前置条件是登录用例执行成功，如果这个用例成功，则由操作者选择其他操作。

(3) 修改密码。管理员登录后，可以修改个人的登录密码，没有登录的用户访问此页面，则直接定位到登录页面。录入的旧密码正确，才允许修改密码。

(4) 查看个人信息。管理员登录后，可以查看管理员的个人信息，没有登录的用户访问此页面，则直接定位到登录页面。

(5) 修改个人信息。管理员登录后，可以修改管理员的个人信息，没有登录的用户访问此页面，则直接定位到登录页面。

7.1.2 角色管理模块

1. 用例图

角色管理模块用例如图 7-2 所示。

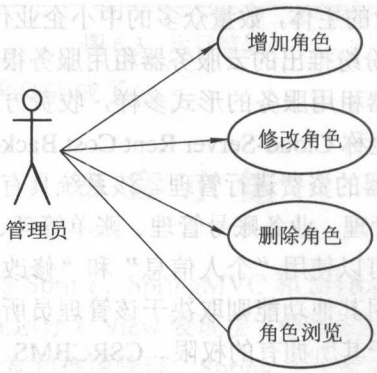

图 7-2 基础信息模块用例图

2. 用例描述

(1) 增加角色。具有“角色管理”权限的管理员登录成功，且“角色浏览”用例执行成功，可以增加新角色，没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“角色管理”权限的管理员访问此页面，则定位到无权访问的错误提示页。注意角色名称唯一，不能重复。

(2) 角色浏览。具有“角色管理”权限的管理员登录后，可以查看所有角色信息。没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“角色管理”权限的管理员访问此页面，则定位到无权访问的错误提示页。

(3) 修改角色。具有“角色管理”权限的管理员登录后，可以修改已有角色信息。

(4) 删除角色。具有“角色管理”权限的管理员登录后，可以删除已有的角色。没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“角色管理”权限的管理员访问此页面，则定位到无权访问的错误提示页。

7.1.3 管理员管理模块

1. 用例图

管理员管理模块用例如图 7-3 所示。

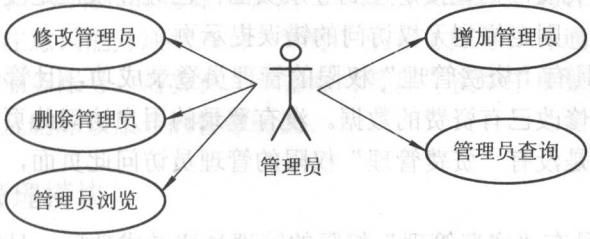

图 7-3 管理员管理模块用例图

2. 用例描述

(1) 增加管理员。具有“管理员管理”权限的管理员登录成功，且“管理员浏览”用例执行成功，可以增加管理员。没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“管理员管理”权限的管理员访问此页面，则定位到无权访问的错误提示页。

(2) 管理员浏览。具有“管理员管理”权限的管理员登录后，可以查看所有管理员信息。

(3) 修改管理员。具有“管理员管理”权限的管理员登录成功，且“管理员浏览”用例执行成功，可以修改已有管理员的数据。没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“管理员管理“权限的管理员访问此页面，则定位到无权访问的错误提示页。

(4) 删除管理员。具有“管理员管理”权限的管理员登录成功，且“管理员浏览”用例或“管理员查询”用例执行成功，可以删除已有的管理员。

(5) 管理员查询。具有“管理员管理”权限的管理员登录成功，且“管理员浏览”用例执行成功，可以查询管理员信息。

7.1.4 资费管理模块

1. 用例图

资费管理模块用例如图 7-4 所示。

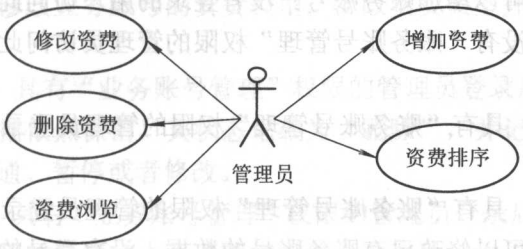

图 7-4 资费管理模块用例图

2. 用例描述

(1) 增加资费。具有“资费管理”权限的管理员登录成功，且“资费浏览”用例或“资

费排序”用例执行成功后，可以增加资费数据。没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“资费管理”权限的管理员访问此页面，则定位到无权访问的错误提示页。

(2) 资费浏览。具有“资费管理”权限的管理员登录成功后，可以浏览资费信息。没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“资费管理”权限的管理员访问此页面，则定位到无权访问的错误提示页。

(3) 修改资费。具有“资费管理”权限的管理员登录成功，且管理员“资费浏览”用例执行成功后，可以修改已有资费的数据。没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“资费管理”权限的管理员访问此页面，则定位到无权访问的错误提示页。

(4) 删除资费。具有“资费管理”权限的管理员登录成功后，且管理员“资费浏览”或“查询资费”用例成功，可以删除暂停状态的资费。

(5) 资费排序。具有“资费管理”权限的管理员登录成功，且“资费浏览”用例执行成功后，可以对资费数据进行排序。

7.1.5 账务账号管理模块

1 用例图

账务账号管理模块用例如图 7-5 所示。

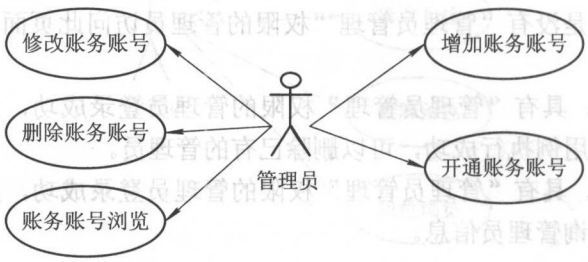

图 7-5 账务账号管理用例图

2. 用例描述

(1) 增加账务账号。具有“账务账号管理”权限的管理员登录成功，且“账务账号浏览”用例执行成功后，可以增加账务账号。没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“账务账号管理”权限的管理员访问此页面，则定位到无权访问的错误提示页。

(2) 账务账号浏览。具有“账务账号管理”权限的管理员登录成功后，可以查看所有账务账号信息。

(3) 修改账务账号。具有“账务账号管理”权限的管理员登录成功，且“账务账号浏览”用例执行成功后，可以修改已有账务账号的数据。没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“账务账号管理”权限的管理员访问此页面，则定位到无权访问的错误提示页；如果没有查询到相应的账务账号数据，则直接返回到的“账务账号”界面。

(4) 删除账务账号。具有“账务账号管理”权限的管理员登录成功，且“账务账号浏览”用例或“账务账号查询”用例执行成功后，可以删除已有的账务账号。账务账号被删除后，数据依然保留，其状态记载为“删除”，并记载删除时间；删除状态的账务账号，不能再开通、暂停或者修改；删除某账务账号，则同时删除其下属的所有业务账号。

(5) 开通账务账号。具有“账务账号管理”权限的管理员登录后，可以开通“暂停”状态下的账务账号。注意只能开通状态为“暂停”的账务账号；开通某账务账号，不会同时开通其下属的所有业务账号，需要在“业务账号管理”模块中单独操作；执行开通操作后，记载开通时间，且删除该账号的暂停时间。

7.1.6　业务账号管理模块

1. 用例图

业务账号管理模块用例如图 7-6 所示。

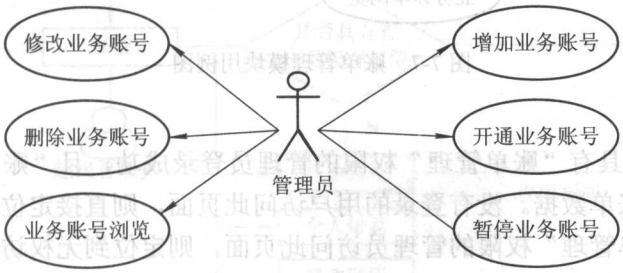

图 7-6　业务账号管理模块用例图

2. 用例描述

(1) 增加业务账号。具有“业务账号管理”权限的管理员登录后，可以增加业务账号。注意新创建的业务账号的状态为开通；状态为“暂停”或者“删除”的账务账号，不能为其添加业务账号；身份证号码和账务账号必须匹配；同一个服务器上的 OS 账号必须唯一。

(2) 业务账号浏览。具有“业务账号管理”权限的管理员登录后，可以查看所有业务账号的信息。没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“业务账号管理”权限的管理员访问此页面，则定位到无权访问的错误提示页。

(3) 修改业务账号。具有“业务账号管理”权限的管理员登录后，可以修改已有业务账号的数据，注意只能修改业务账号的资费类型；修改资费类型并保存成功后，新的资费标准从下月生效。

(4) 删除业务账号。具有“业务账号管理”权限的管理员登录后，可以删除业务账号。业务账号被删除后，数据依然保留，其状态记载为“删除”，并记载删除时间；删除状态的业务账号，不能再开通、暂停或者修改。

(5) 开通业务账号。具有“业务账号管理”权限的管理员登录后，可以开通”暂停“状态下的业务账号。注意只能开通状态为“暂停“的业务账号；如果业务账号所属的账务账号的状态为暂停或者删除，则不能开通此业务账号；执行开通操作后，记载开通时间，且删除该账号的暂停时间。

(6) 暂停业务账号。具有“业务账号管理”权限的管理员登录后，可以暂停“开通”

状态下的业务账号。注意只能暂停状态为“开通”的业务账号；执行暂停操作时，需要记载该账号的暂停时间。

7.1.7 账单管理模块

1. 用例图

账单管理模块用例如图 7-7 所示。

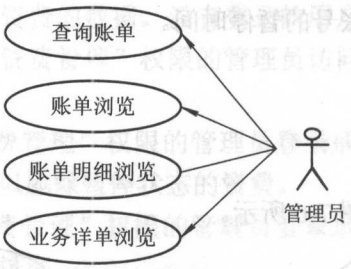

图 7-7 账单管理模块用例图

2. 用例描述

(1) 查询账单。具有“账单管理”权限的管理员登录成功，且“账单浏览”用例执行成功后，可以查询账单数据。没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“账单管理”权限的管理员访问此页面，则定位到无权访问的错误提示页；注意只能查询近 3 年的账单，即只查询当前年及其前两年的账单数据。

(2) 账单浏览。具有“账单管理”权限的管理员登录后，可以查看所有账单的信息。没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“账单管理”权限的管理员访问此页面，则定位到无权访问的错误提示页；当月账单不显示；账单的支付状态有两种：已支付和未支付；账单的支付方式有三种：现金、转账、卡付。

(3) 账单明细浏览。具有“账单管理”权限的管理员登录后，可以查看某账单的明细数据。此用例执行成功，可以执行“业务详单浏览”用例。

(4) 业务账单详情。具有“账单管理”权限的管理员登录后，可以查看某业务账号的业务详单，没有登录的用户访问此页面，则直接定位到登录页面；已经登录但是没有“账单管理”权限的管理员访问此页面，则定位到无权访问的错误提示页。

7.1.8 报表模块

1. 用例图

报表模块的用例如图 7-8 所示。

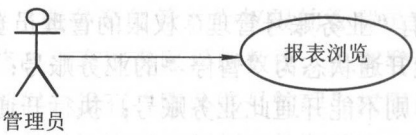

图 7-8 报表模块用例图

2. 用例描述

报表浏览：具有“报表”权限的管理员登录后，可以查看所有报表的信息。

7.2 概要设计

7.2.1 系统流程

管理员登录系统后，如果该管理员具有系统管理权限，则可以创建角色和分配权限，添加其他管理员账号，也可以对系统其他功能模块进行操作，如果是非超级管理员成功登录，只能操作超级管理员分配的权限对应的功能模块，如图 7-9 所示。

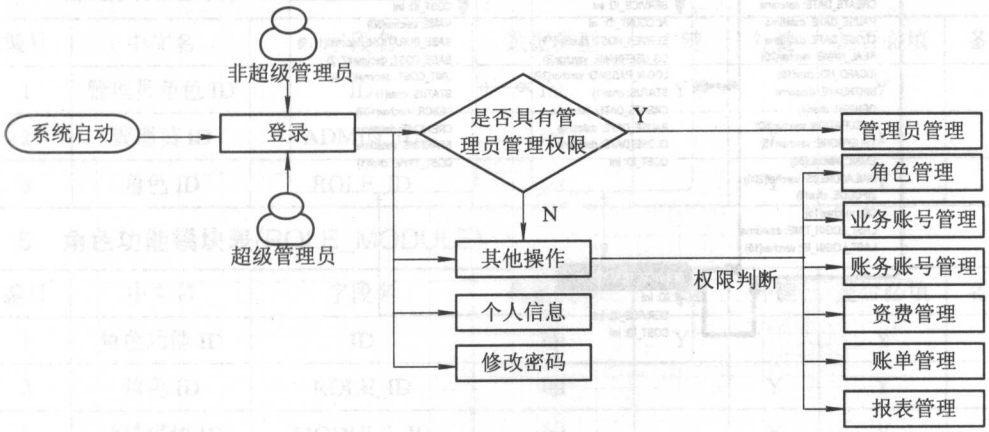

图 7-9 系统流程图

7.2.2 功能模块图

该系统具有的功能主要有基础信息、角色管理、管理员管理、资费管理、账务管理、业务管理、账单管理以及报表管理，如图 7-10 所示。

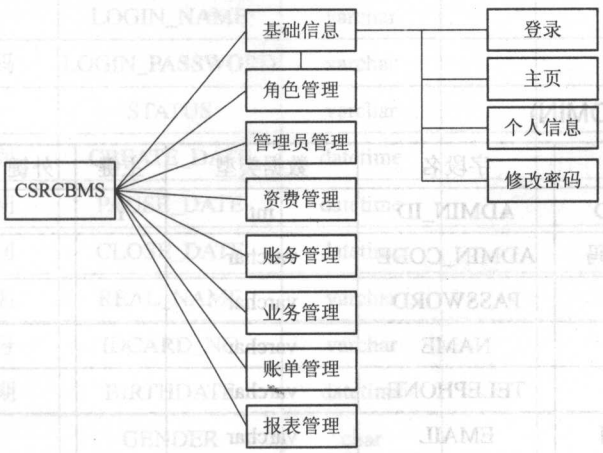

图 7-10 系统功能模块图

7.3 数据库设计

7.3.1 数据模型

该系统数据模型包括账号表、业务表、资费表、业务更新表、模块表、角色模块表、角色表、管理员角色表、管理员表，它们之间的关系如图 7-11 所示。

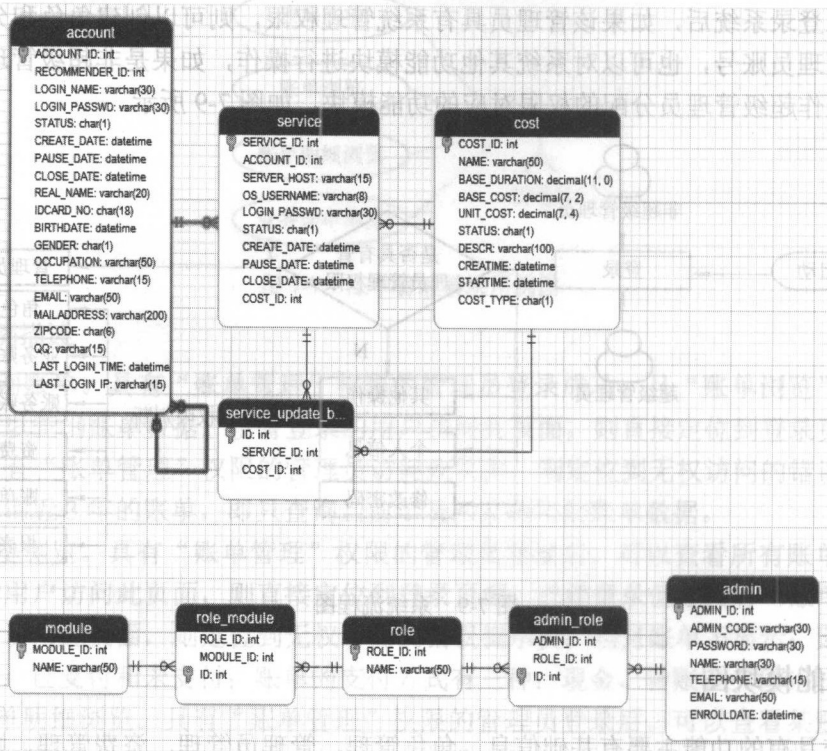

图 7-11　系统数据模型

7.3.2 数据字典

1. 管理员表(ADMIN)

编号	中文名	字段名	数据类型	主键	外键	是否必填	备注
1	管理员 ID	ADMIN_ID	int	Y		Y	
2	管理员编码	ADMIN_CODE	varchar			Y	
3	密码	PASSWORD	varchar			Y	
4	姓名	NAME	varchar				
5	电话	TELEPHONE	varchar				
6	电子邮箱	EMAIL	varchar				
7	登记日期	ENROLLDATE	datetime			Y	

2. 角色表(ROLE)

编号	中文名	字段名	数据类型	主键	外键	是否必填	备注
1	角色 ID	ROLE_ID	int	Y		Y	
2	角色名	ROLE_NAME	varchar			Y	

3. 功能模块表(MODULE)

编号	中文名	字段名	数据类型	主键	外键	是否必填	备注
1	模块 ID	MODULE_ID	int	Y		Y	
2	模块名	MODULE_NAME	varchar			Y	

4. 管理员角色表(ADMIN_ROLE)

编号	中文名	字段名	数据类型	主键	外键	是否必填	备注
1	管理员角色 ID	ID	int	Y		Y	
2	管理员 ID	ADMIN_ID	int		Y	Y	
3	角色 ID	ROLE_ID	int		Y	Y	

5. 角色功能模块表(ROLE_MODULE)

编号	中文名	字段名	数据类型	主键	外键	是否必填	备注
1	角色功能 ID	ID	int	Y		Y	
2	角色 ID	ROLE_ID	int		Y	Y	
3	功能模块 ID	MODULE_ID	int		Y	Y	

6. 账号账务表(ACCOUNT)

编号	中文名	字段名	数据类型	主键	外键	是否必填	备注
1	账号账务 ID	ACCOUNT_ID	int	Y		Y	
2	推荐人 ID	RECOMMENDER_ID	int		Y		
3	登录名	LOGIN_NAME	varchar			Y	
4	登录密码	LOGIN_PASSWORD	varchar			Y	
5	状态	STATUS	varchar			Y	
6	创建时间	CREATE_DATE	datetime			Y	
7	暂停时间	PAUSE_DATE	datetime				
8	关闭时间	CLOSE_DATE	datetime				
9	真实姓名	REAL_NAME	varchar			Y	
10	身份证号	IDCARD_NO	varchar			Y	
11	出生日期	BIRTHDATE	datetime				
12	性别	GENDER	char				
13	职业	OCCUPATION	varchar				

续表

编号	中文名	字段名	数据类型	主键	外键	是否必填	备注
14	电话	TELEPHONE	varchar			Y	
15	电子邮箱	EMAIL	varchar				
16	联系地址	MAILADDRESS	varchar				
17	邮编号码	ZIPCODE	varchar				
18	QQ 号码	QQ	varchar				
19	最后登录时间	LAST_LOGIN_TIME	datetime				
20	最后登录 IP	LAST_LOGIN_IP	datetime				

7. 服务账务表(SERVICE)

编号	中文名	字段名	数据类型	主键	外键	是否必填	备注
1	服务账务 ID	SERVICE_ID	int	Y		Y	
2	账号账务 ID	ACCOUNT_ID	int		Y		
3	服务器 IP	SERVER_HOST	varchar			Y	
4	用户名	OS_USERNAME	varchar			Y	
5	登录密码	LOGIN_PASSWD	varchar			Y	
6	服务账号的状态	STATUS	char			Y	
7	创建日期	CREATE_DATE	datetime			Y	
8	暂停时间	PAUSE_DATE	datetime				
9	关闭时间	CLOSE_DATE	datetime				
10	资费 ID	COST_ID	int		Y	Y	

8. 资费表(COST)

编号	中文名	字段名	数据类型	主键	外键	是否必填	备注
1	资费 ID	COST_ID	int	Y		Y	
2	资费名	NAME	varchar			Y	
3	基本时长	BASE_DURATION	varchar			Y	
4	基本费用	BASE_COST	varchar			Y	
5	单位费用	UNIT_COST	varchar				
6	状态	STATUS	char			Y	
7	资费说明	DESCR	datetime			Y	
8	创建时间	CREATIME	datetime			Y	
9	开启时间	STARTIME	datetime				
10	资费类型	COST_TYPE	char			Y	

9. 服务资费更新表(SERVICE_UPDATE_BAK)

编号	中文名	字段名	数据类型	主键	外键	是否必填	备注
1	服务资费更新 ID	ID	int	Y		Y	
2	服务 ID	SERVICE_ID	int		Y	Y	
3	资费 ID	COST_ID	int		Y	Y	

7.4 功能实现

本书讲解了当前流行的 Java EE 框架技术 SpringMVC、Spring 和 MyBatic，本章节利用前面章节讲述的三大框架系统进行后台实现，不会实现前台页面，本书提供前台页面实现源码仅供参考。为了降低项目实践的难度，本系统不涉及业务逻辑层，但并不影响三大框架系统知识体系的综合应用和项目的实用性，本系统的功能采用由下至上的开发思想来实现，即按照实体层、映射层、数据访问层和控制层的开发顺序实现系统的功能。

7.4.1 基础信息模块实现

该模块包括查看个人信息、修改个人信息、修改密码和编写系统主页，涉及的数据库表都是单表操作，每个功能点比较简单，实现容易，所以该模块仅实现登录功能。

(1) 用户登录系统成功后会将用户信息保存在会话中，与用户信息相对应的实体类为 Admin，其实现的主要代码如下：

```
public class Admin {
    private Integer admin_id;//管理员Id
    private String adminCode;//管理员账号
    private String password;//密码
    private String name;//名称
    private String telephone;//电话号码
    private String email;//邮编
    private Timestamp enrollDate;//注册时间
    private List<Role> roles;//角色列表
    private List<Integer> roleIds;//角色Id列表
    //getter和setter方法略
}
```

上面代码列出了 Admin 类的属性和对应的 getter 和 setter 方法。有一个集合类型 roles 属性，表示用户分配的角色，它的作用为检索登录用户可以操作的系统模块。需要特别说明一下，之后贴出的实体类的源代码，属性的 getter 和 setter 方法不再贴出。

(2) 用户登录需要查询数据库，所以需要创建查询 SQL 语句，并发送 SQL 语句实现数

据的查询，然后将查询的结果返回。SQL 语句的定义在 SQL 映射文件 AdminMapper.xml 实现，其代码如下：

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//ibatis.apache.org//DTD Mapper 3.0//EN"
 "http://ibatis.apache.org/dtd/ibatis-3-mapper.dtd">
<mapper namespace = "com.cscbms.dao.AdminDao">
    <select id = "findByCode"
        parameterType = "string"
        resultType = "com.cscbms.entity.Admin">
        select * from admin where admincode = #{adminCode}
    </select>
</mapper>
```

上面代码<mapper>元素的 namespace 设置了映射接口为 AdminDao。<select>元素定义查询语句，id 属性指命名空间中唯一的标识符 findByCode；parameterType 指将传入这条语句的参数的完全限定名和别名；resultType 指从这条语句中返回的期望类的完全限定名和别名。

(3) 在映射接口 AdminDao 中添加接口方法 findByCode，方法的参数为 String 类型，返回类型为 Admin。注意 AdminDao 接口名和 AdminMapper.xml 中的 mapper 属性 namespace 相同，findByPage 方法签名和 AdminMapper.xml 中 id 为 findByPage 的 select 语句相匹配。AdminDao 代码如下：

```
@MyBatisRepository
public interface AdminDao {
    //List<Admin> findByPage(Page page);
    //int findRows(Page page);
    //void updatePassword(Map<String, Object> param);
    //Admin findById(int id);
    //void saveAdmin(Admin admin);
    //void saveAdminRoles(Map<String, Object> adminRoles);
    //void updateAdmin(Admin admin);
    //void deleteAdminRoles(int admin_id);
    //void deleteAdmin(int id);
    //根据管理员账号查询
    Admin findByCode(String adminCode);
    //List<Module> findModulesByAdmin(int admin_id);
}
```

上面代码定义了根据管理员账号查询的接口方法，返回 Admin 对象。该方法用于验证登录用户是否是合法用户。

(4) 映射接口 AdminDao 的实现，根据映射接口自动生成接口的实现类对象，在

applicationContext.xml 中添加一个 MapperScannerConfigurer 类型 bean，实现自动扫映射接口包。代码如下：

```
<bean class = "org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name = "basePackage" value = "com.cscbms.dao" />
        <property name = "annotationClass"
        value = "com.cscbms.annotation.MyBatisRepository"/>
    </bean>
```

上面代码表示在 Spring 容器中注册一个 MapperScannerConfigurer 对象，该对象属性 basePackage 设置为 com.cscbms.dao，属性 annotationClass 设置为 MyBatisRepository，表示 Spring 自动扫描 com.cscbms.dao 和子包，如果在 com.cscbms.dao 包和子包中的接口有 MyBatisRepository 注解，Spring 会自动生成该接口的实现类对象，并由 spring 容器管理。

(5) 创建 LoginController 类，在类级别添加@Controller 和@RequestMapping，在该类添加 login()方法，处理登录页面传到后台服务器的信息；添加 createImage()方法产生验证码。代码如下：

```
@Controller //注册为 Spring 容器 bean
@RequestMapping("/login") //请求映射
public class LoginController extends BaseController {
    @Resource //注入 adminDao 对象
    private AdminDao adminDao;
    //请求映射，当请求包含/login/login.do 时，调用该方法
    @RequestMapping("/login.do")
    @ResponseBody
    public Map<String, Object> login(
            String adminCode,
            String password,
            String code,
            HttpSession session) {
        Map<String, Object> result = new HashMap<String, Object>();
        String imageCode = (String) session.getAttribute("imageCode");
        if(code == null
                || !code.equalsIgnoreCase(imageCode))
        {
            result.put("flag", IMAGE_CODE_ERROR);
            return result;
        }
        Admin admin = adminDao.findByCode(adminCode);
        if(admin == null)
        {
```

```
                result.put("flag", ADMIN_CODE_ERROR);
                return result;
            }
            else if (!admin.getPassword().equals(password))
            {
                result.put("flag", PASSWORD_ERROR);
                return result;
            }
            else
            {
                session.setAttribute("admin", admin);
                List<Module> modules =
                    adminDao.findModulesByAdmin(admin.getAdmin_id());
                System.out.println(modules.size());
                session.setAttribute("allModules", modules);
                result.put("flag", SUCCESS);
                return result;
            }
        }
    }
}
```

在 LoginController 类中，为了系统安全，添加验证码功能，添加了 createImage()方法用于生成登录页面的验证码，当请求包含/login/createImage.do 时就会调用该方法返回生成的验证码，代码如下：

```
/**
 * 产生登录页面验证码
 */
@RequestMapping("/createImage.do")
public void createImage(
            HttpServletResponse response, HttpSession session)
            throws Exception {
    Map<String, BufferedImage> imageMap = ImageUtil.createImage();
    String code = imageMap.keySet().iterator().next();
    session.setAttribute("imageCode", code);
    BufferedImage image = imageMap.get(code);
    response.setContentType("image/jpeg");
    OutputStream ops = response.getOutputStream();
    ImageIO.write(image, "jpeg", ops);
```

```
        ops.close();
    }
```

还添加一个 toIndex()方法，当管理员用户登录成功，就可以调用该方法，返回“main/index”。经过资源视图解析器将“main/index”组装为/WEB-INF/index.jsp，并跳转到该页面如下：

```
//请求映射，当请求包含/login/toIndex.do 时，调用该方法
    @RequestMapping("/toIndex.do")
    public String toIndex() {
        return "main/index";
    }
```

(6) 系统用户登录界面。

未登录用户访问该系统，呈现给用户的是登录界面，要求用户输入用户名、密码和验证码，如图 7-12 所示。

该登录页面会对用户输入的数据进行合法性验证，前台验证通过就将用户信息提交后台服务器处理。如果用户名、密码和验证码都输入正确，就跳转到系统主页；否则给出错误提示，错误提示分三种情况，分别是账号错误、密码错误和验证码错误。

图 7-12　系统用户登录界面

7.4.2　角色管理功能实现

超级管理员登录系统后，添加角色，以便对不同管理员分配角色，不同的角色拥有相

应的系统模块操作权限。角色管理模块包括查询角色、添加角色、修改角色和删除角色，下面对角色模块的功能进行实现。

1) 实体 Role 类的实现

通常实体类的成员变量和数据库角色表的字段一一对应，且成员变量名和表的字段名应尽量一致，实体 Role 与数据库表 role 对应，基于这种约定，Role 类代码实现如下：

```
public class Role {
    private Integer roleId;
    private String name;
    private List<Module> modules;
    private List<Integer> moduleIds;
//属性对应的 getter 和 setter 方法略
}
```

在 Role 类的成员变量中有 Module 和 ModuleId 的集合，指角色拥有的可以操作系统模块集合。

2) 创建映射接口 RoleDAO

该接口定义对数据库增加角色、删除角色、修改角色和查询角色及角色模块的操作等方法。

(1) 在 RoleDao 映射接口中，角色的查找有多种方法，可以根据 Page 对象、角色 id、角色名称查找，代码如下所示：

```
@MyBatisRepository
public interface RoleDao {
//根据角色Id来查找Role对象
Role findRoleById(int id);
//根据角色名查询Role
Role findRoleByName(String name);
//根据page查找角色对象，返回Role类的列表
List<Role> findByPage(Page page);
}
```

(2) RoleDao 接口提供新增、修改和删除角色的方法，代码如下：

```
void saveRole(Role role); //保存角色对象
void updateRole(Role role); //更新角色
void deleteRole(int roleId);//删除Role对象根据角色Id
```

(3) RoleDao 也提供了操作模块对象的接口。代码如下：

```
//根据角色Id查询模块列表
List<Module> findModules(int roleId)
//根据角色Id删除RoleModules对象
void deleteRoleModules(int roleId);
```

```
//保存角色模块对象
void saveRoleModules(Map<String, Object> roleModules);
```

3) 创建 RoleMapper.xml 角色映射 xml 文件

定义 SQL 语句与 RoleDao 映射接口方法相匹配，设置 mapper 元素的 namespace 属性为 RoleDao，将映射接口和映射文件相关联。在 mapper 元素内添加 SQL 语句。代码如下：

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//ibatis.apache.org//DTD Mapper 3.0//EN"
 "http://ibatis.apache.org/dtd/ibatis-3-mapper.dtd">
<mapper namespace = "com.zds.dao.RoleDao">
</mapper>
```

(1) 在 RoleMapper.xml 接口添加查询语句。定义一个 SQL 语句，通常要设置 id、参数类型和返回类型。代码如下：

```
<select id = "findByName" parameterType = "string" resultMap = "roleMap">
    select * from role_info where name = #{name}
</select>
<select id = "findRoleById" parameterType = "int" resultMap = "roleMap">
    select * from role_info where roleid = #{id}
</select>
<select id = "findRows"
    parameterType = "com.zds.entity.page.Page"
    resultType = "int">
    select count(*) from role_info
</select>
<select id = "findByPage"
    parameterType = "com.zds.entity.page.Page"
    resultMap = "roleMap">
<![CDATA[
        select * from role_info order by roleid limit #{begin}, #{end}
    ]]>
</select>
```

在条件查询设置字段值时，用#{参数名}获取参数值，如果参数名为 id，可以写成#{id}；如果参数是 Page 类 page 对象，Page 类有 begin、end 属性，可以用#{begin}、#{end}获取 page 的 begin、end 的值。id 为“findRows”的查询语句查询所有角色记录数。

(2) 为了对角色对象进行新增、修改和删除操作，在 RoleMapper.xml 中定义相应 SQL 语句，代码如下：

```
//添加角色记录
<insert id = "saveRole" parameterType = "com.zds.entity.Role">
    insert into role(name) values(
```

```
    #{name, jdbcType = VARCHAR}
    )
</insert>
//修改角色记录
<update id = "updateRole" parameterType = "com.zds.entity.Role">
    update role set name = #{name} where role_id = #{ roleId }
</update>
//删除角色记录
<delete id = "deleteRole" parameterType = "int">
    delete from role where role_id = #{roleId}
</delete>
```

在上面代码中定义一个持久化 Role 对象的语句，定义根据角色 Id 修改和删除角色对象的 SQL 的语句。

为了能支持角色添加和删除操作模块，在RoleMapper.xml文件中添加相应的SQL语句。代码如下：

```
<insert id = "saveRoleModules" parameterType = "hashMap">
    insert into role_module values(
    #{roleId, jdbcType = NUMERIC},
    #{moduleId, jdbcType = NUMERIC}
    )
</insert>
<delete id = "deleteRoleModules" parameterType = "int">
    delete from role_module where roleId = #{roleId}
</delete>
```

上面代码定义在角色模块表中添加记录语句和根据角色Id在角色模块表中删除记录的语句。

4) 在 Spring 容器中创建 RoleController 对象

在类级别上添加@RequestMapping 注解，设置属性 value 的值为“/role”，SpringMVC 框架将请求路径中包含“/role”的请求映射到 RoleController 对象，并由 RoleController 对象处理该请求；在类级别上@SessionAttributes，声明 Session 属性；定义成员变量 RoleDao 对象，在 RoleDao 对象上添加@Resource，SpringMVC 会自动注入 RoleDao 对象，利用 RoleDao 对象的接口方法对角色对象实现持久化操作。代码如下：

```
@Controller
@RequestMapping("/role")
@SessionAttributes("rolePage")
public class RoleController {
    @Resource
    private RoleDao roleDao;
```

```
}
```

(1) 在 RoleController 类中添加保存 Role 对象的方法，代码如下：

```
@RequestMapping("/addRole.do")
public String add(Role role, Model model) {
    roleDao.saveRole(role);
    List<Integer> moduleIds = role.getModuleIds();
    for (Integer moduleId : moduleIds) {
        Map<String, Object> roleModules =
                new HashMap<String, Object>();
        roleModules.put("role_id", role.getRole_id());
        roleModules.put("module_id", moduleId);
        roleDao.saveRoleModules(roleModules);
    }
    return "redirect:findRole.do";
}
```

在上面代码中@RequestMapping 注解设置请求映射，解决该方法处理什么请求的问题。在添加 Role 对象时，调用 RoleDao 的方法来具体实现，在添加 Role 对象的同时也为角色对象分配了模块，所以角色和模块间的关系需要持久化，即在 role_module 表中添加记录，将该角色可以操作的模块进行保存。

(2) 在 RoleController 类中添加查询 Role 对象的方法，代码如下：

```
@RequestMapping("/findRole.do")
public String find(RolePage page, Model model) {
    page.setRows(roleDao.findRows(page));
    model.addAttribute("rolePage", page);
    List<Role> roles = roleDao.findByPage(page);
    model.addAttribute("roles", roles);
    return "role/role_list";
}
```

在设计 find()方法时，添加了 RolePage 和 Model 类型的两个参数，RolePage 对象封装查询对象，Model 对象将查询结果作为属性保存以便在前台页面展示查询结果。

(3) 在 RoleController 类中添加修改 Role 对象的方法，代码如下：

```
@RequestMapping("/updateRole.do")
public String update(Role role, Model model) {
    roleDao.updateRole(role);//更新角色对象
    //删除根据角色编号删除角色模块
    roleDao.deleteRoleModules(role.getRoleId());
    //获取修改后的该角色操作的模块Ids集合
    List<Integer> moduleIds = role.getModuleIds();
```

```
        for (Integer moduleId : moduleIds) {
            Map<String, Object> roleModules =
                    new HashMap<String, Object>();
            roleModules.put("roleId", role.getRoleId());
            roleModules.put("moduleId", moduleId);
            roleDao.saveRoleModules(roleModules);
        }
        //跳转到findRole.jsp页面
        return "redirect:findRole.do";
    }
```

上面代码实现了 Role 对象的更新功能，在修改角色对象时，可能修改了角色可以操作的模块，所以为了保存这个变化，我们需要在 role_module 中删除角色可以操作的模块记录，然后再保存修改后角色模块记录，最后将请求跳转到 findRole.jsp 页面。

(4) 在 RoleController 类中添加删除 Role 对象的方法，代码如下：

```
@RequestMapping("/deleteRole.do")
    public String delete(@RequestParam("id") int id) {
        roleDao.deleteRoleModules(id);
        roleDao.deleteRole(id);
        return "redirect:findRole.do";
    }
```

删除方法很简单，首先根据角色 Id 删除 role_module 表的记录，然后删除 role 表中的对应记录。操作成功后将请求转发到 findRole.jsp 页面。

5) 角色管理模块的主要用户界面

因删除功能没有对应的页面，所以没有给出角色管理模块的主要用户界面。

(1) 查询角色的用户界面如图 7-13 所示。

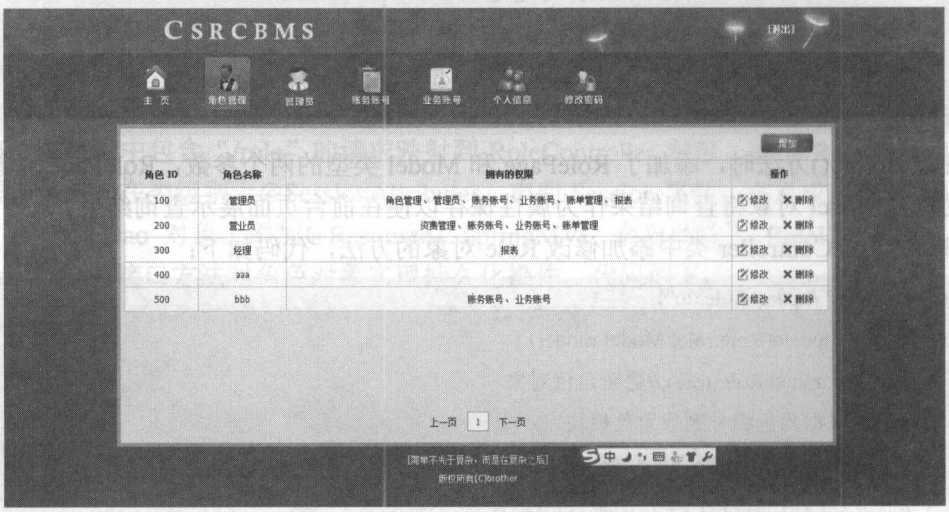

图 7-13　查询角色

(2) 添加角色的用户界面如图 7-14 所示。

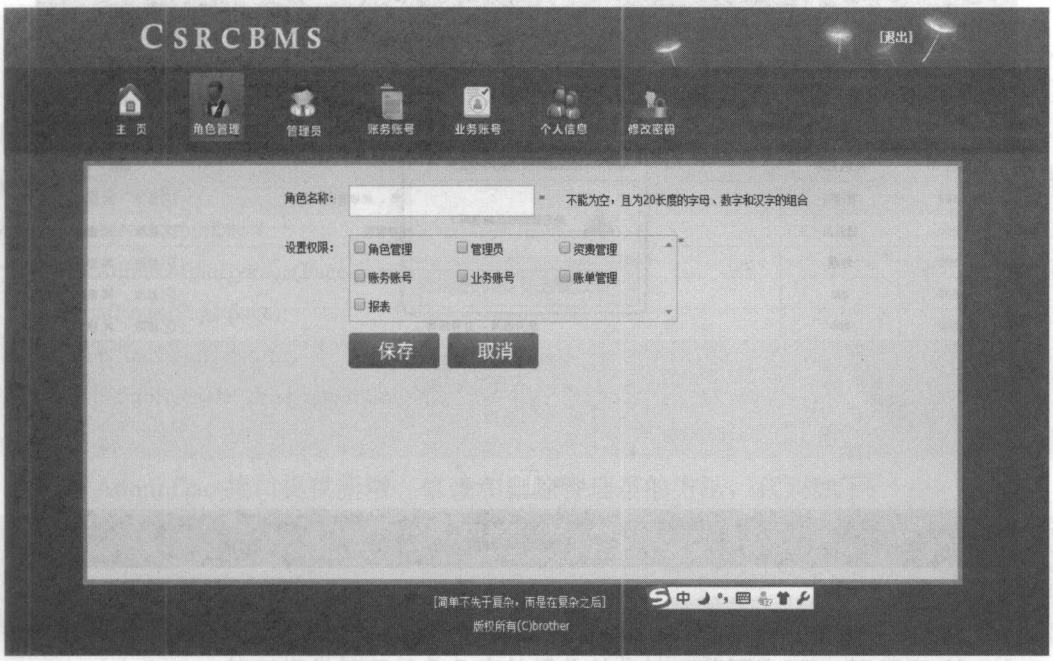

图 7-14 添加角色

(3) 修改角色的用户界面如图 7-15 所示。

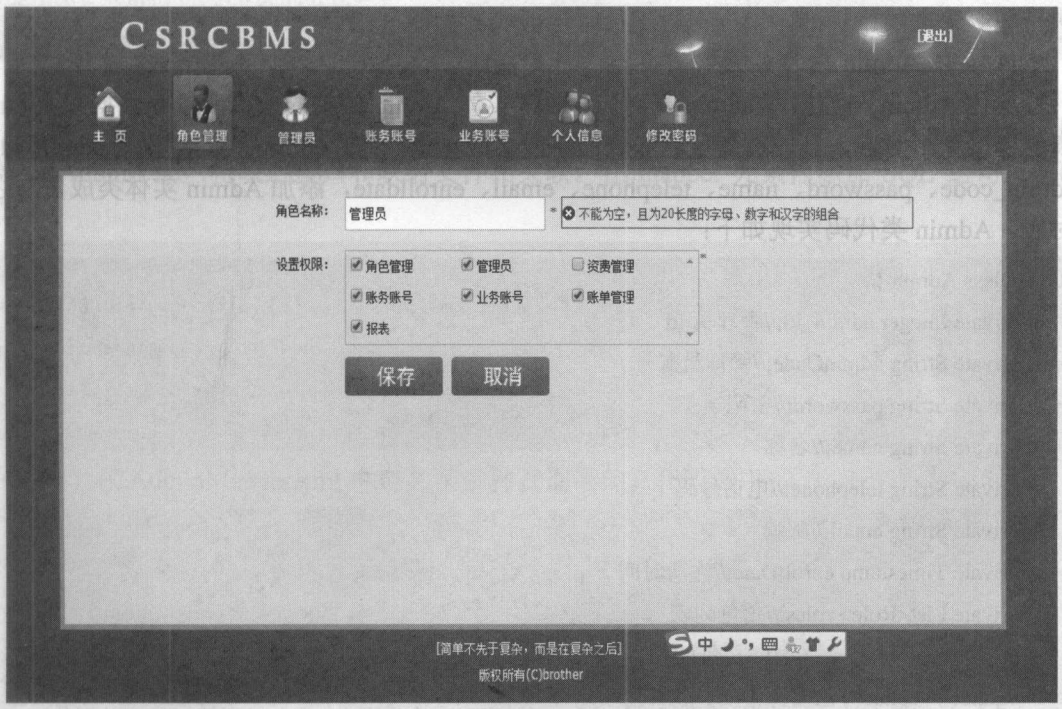

图 7-15 修改角色

(4) 删除角色的用户界面如图 7-16 所示。

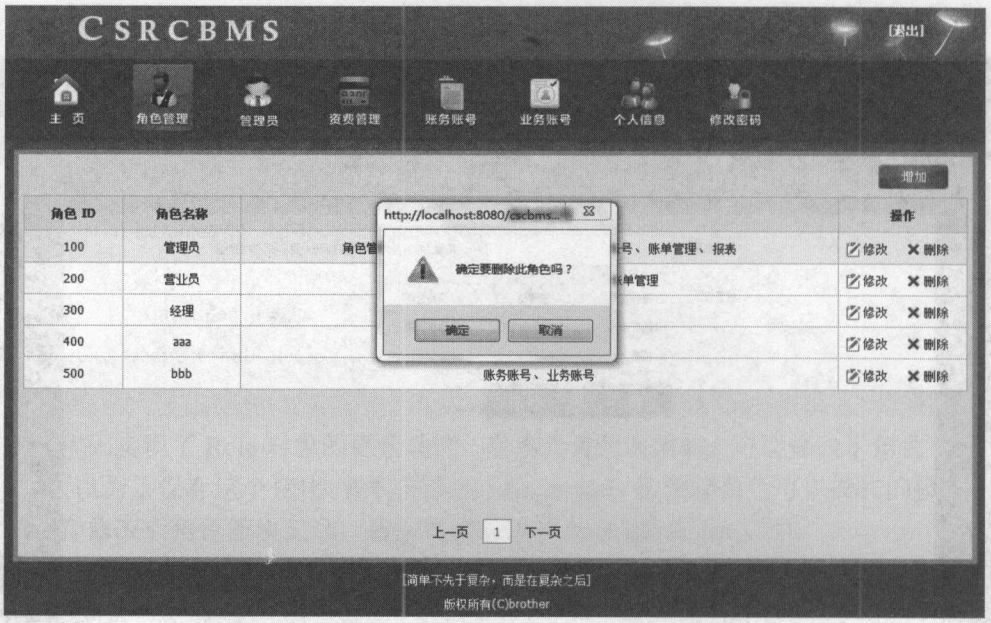

图 7-16　删除角色

7.4.3　管理员管理功能实现

管理员管理模块包括管理员查询、管理员添加、管理员修改和管理员删除功能。此模块是超级管理员维护的。下面对管理员管理模块的功能进行实现。

1) 实体 Admin 类的实现

实体 Admin 与数据库 admin 表对应，通常实体 Admin 层类的成员变量和数据库 admin 表的字段一一对应，且成员变量名和表的字段名应尽量一致，admin 表的字段有 admin_id、admin_code、password、name、telephone、email、enrolldate，添加 Admin 实体类成员与之对应。Admin 类代码实现如下：

```
public class Admin {
    private Integer admin_id;//管理员 Id
    private String adminCode;//管理员账号
    private String password;//密码
    private String name;//名称
    private String telephone;//电话号码
    private String email;//邮编
    private Timestamp enrollDate;//注册时间
    private List<Role> roles;//角色列表
    private List<Integer> roleIds;//角色 Id 列表
    //getter 和 setter 方法略
}
```

为了获取 Admin 对象属于的角色，在 Admin 中添加成员变量 roles，通过 Admin 对象

getRoles()方法获取该对象属于的角色集合，然后通过角色获取操作模块列表。

2) 创建映射接口 AdminDAO

该接口定义增加管理员、删除管理员、修改管理员和查询管理员等方法。

(1) 在 AdminDAO 映射接口中，提供多种查找管理员的方法，可以根据 Page 对象、管理员 id、管理员编码角色名称查找。代码如下：

```
@MyBatisRepository
public interface AdminDao {
    List<Admin> findByPage(Page page);
    int findRows(Page page);
    Admin findById(int id);
    Admin findByCode(String adminCode);
}
```

(2) AdminDao 接口提供新增、修改和删除管理员的方法，代码如下：

```
void saveAdmin(Admin admin);
void updateAdmin(Admin admin);
void deleteAdmin(int id);
```

(3) AdminDao 接口提供管理员角色表的操作的方法，代码如下：

```
void saveAdminRoles(Map<String, Object> adminRoles);
void deleteAdminRoles(int adminId);
List<Module> findModulesByAdmin(int adminId);
```

3) 创建映射文件 AdminMapper.xml

定义与 RoleDao 映射接口方法匹相配的 SQL 语句，设置 mapper 元素的 namespace 属性为 AdminDao，将映射接口和映射文件相关联。在 mapper 元素内添加 SQL 语句。代码如下：

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//ibatis.apache.org//DTD Mapper 3.0//EN"
 "http://ibatis.apache.org/dtd/ibatis-3-mapper.dtd">
<mapper namespace = "com.zds.dao.AdminDao">
</mapper>
```

(1) 在 AdminMapper.xml 中定义了多种查询语句。代码如下：

```
<select id = "findByPage"
        parameterType = "com.cscbms.entity.page.Page"
        resultMap = "adminMap">
            select * from admin where admin_id in (
            select a.admin_id
            from admin a
            left join admin_role ar on a.admin_id = ar.admin_id
```

```
            left join role_info ri on ri.role_id = ar.role_id
            left join role_module rm on rm.role_id = ri.role_id
            <where>
                <if test = "roleName != null &amp;amp;&amp;amp; roleName.length()>0">
                    and ri.name like '%'||#{roleName}||'%'
                </if>
                <if test = "moduleId != null">
                    and rm.module_Id = #{moduleId}
                </if>
            </where>
        ) order by admin_id limit #{begin}, #{end}

    </select>
<select id = "findById" parameterType = "int" resultMap = "adminMap">
        select * from admin where admin_id = #{id}
    </select>
<select id = "findByCode"
        parameterType = "string"
        resultType = "com.cscbms.entity.Admin">
        select * from admin where admin_code = #{adminCode}
    </select>
```

(2) 在 AdminMapper.xml 中定义修改、删除和添加 admin 表记录的语句。代码如下：

```
<insert id = "saveAdmin" parameterType = "com.cscbms.entity.Admin">
        insert into admin(admin_code, password, name
            telephone, email, enrolldate) values(
            #{adminCode, jdbcType = VARCHAR},
            #{password, jdbcType = VARCHAR},
            #{name, jdbcType = VARCHAR},
            #{telephone, jdbcType = VARCHAR},
            #{email, jdbcType = VARCHAR},
            #{enrollDate, jdbcType = TIMESTAMP}
        )
    </insert>
<update id = "updateAdmin" parameterType = "com.cscbms.entity.Admin">
        update admin set
            name = #{name, jdbcType = VARCHAR},
            telephone = #{telephone, jdbcType = VARCHAR},
            email = #{email, jdbcType = VARCHAR}
```

```
        where admin_id = #{adminId}
    </update>
<delete id = "deleteAdmin" parameterType = "int">
        delete from admin where admin_id = #{id}
    </delete>
```

(3) 为了给管理员用户分配权限，在 AdminMapper.xml 中定义操作 admin_role 表的 SQL 语句：

```
<insert id = "saveAdminRoles" parameterType = "hashMap">
        insert into adminrole values(
            #{adminId },
            #{roleId }
        )
</insert>
<delete id = "deleteAdminRoles" parameterType = "int">
        delete from admin_role where admin_id = #{adminId}
</delete>
```

4) 创建 AdminCotroller 类

在类级别上添加@Controller 注解，当 Spring 自动扫描到类中@Controller 注解时，在 Spring 容器中创建 AdminCotroller 对象。在类级别上添加@RequestMapping 注解，设置属性 value 的值为“/admin”，SpringMVC 框架将请求路径中包含“/admin”的请求映射到 AdminController 对象，并由 AdminController 对象处理该请求；在类级别上添加@SessionAttributes，声明 Session 属性；定义成员变量 AdminDao 对象，在 AdminDao 对象上添加@Resource，SpringMVC 会自动注入 AdminDao 对象，利用 AdminDao 对象的接口方法对管理员对象实现持久化操作，定义成员变量 AdminDao，获取管理员列表。代码如下：

```
@Controller
@RequestMapping("/admin")
@SessionAttributes("adminPage")
public class AdminController {
    @Resource
    private AdminDao adminDao;
    @Resource
    private RoleDao roleDao;
}
```

(1) 在 AdminController 类中添加查询 Admin 对象的方法，代码如下：

```
@RequestMapping("/findAdmin.do")
    public String find(AdminPage page, Model model) {
        page.setRows(adminDao.findRows(page));
```

```
        model.addAttribute("adminPage", page);
        List<Admin> admins = adminDao.findByPage(page);
        model.addAttribute("admins", admins);
        List<Module> modules = roleDao.findAllModules();
        model.addAttribute("modules", modules);
        return "admin/adminList";
    }
```

(2) 在 AdminController 类中添加添加 Admin 对象的方法，代码如下：

```
@RequestMapping("/addAdmin.do")
public String add(Admin admin, Model model) {
        admin.setEnrollDate(
                new Timestamp(System.currentTimeMillis()));
        adminDao.saveAdmin(admin);
        List<Integer> roleIds = admin.getRoleIds();
        for (Integer roleId : roleIds) {
                Map<String, Object> adminRoles =
                        new HashMap<String, Object>();
                adminRoles.put("adminId", admin.getAdminId());
                adminRoles.put("roleId", roleId);
                adminDao.saveAdminRoles(adminRoles);            }
        return "redirect:findAdmin.do";
}
```

(3) 在 AdminController 类中添加修改 Admin 对象的方法，代码如下：

```
@RequestMapping("/updateAdmin.do")
public String update(Admin admin, Model model) {
        adminDao.updateAdmin(admin);
        adminDao.deleteAdminRoles(admin.getAdminId());
        List<Integer> roleIds = admin.getRoleIds();
        for (Integer roleId : roleIds) {
                Map<String, Object> adminRoles =
                        new HashMap<String, Object>();
                adminRoles.put("adminId", admin.getAdminId());
                adminRoles.put("roleId", roleId);
                adminDao.saveAdminRoles(adminRoles);
        }
        return "redirect:findAdmin.do";
}
```

(4) 在 AdminController 类中添加删除 Admin 对象的方法，代码如下：

```
@RequestMapping("/deleteAdmin.do")
    public String delete(@RequestParam("adminId") int id) {
        adminDao.deleteAdminRoles(id);
        adminDao.deleteAdmin(id);
        return "redirect:findAdmin.do";
    }
```

5) 管理员管理模块的主要用户界面

(1) 查询管理员的用户界面如图 7-17 所示。

图 7-17　查询管理员

(2) 添加管理员的用户界面如图 7-18 所示。

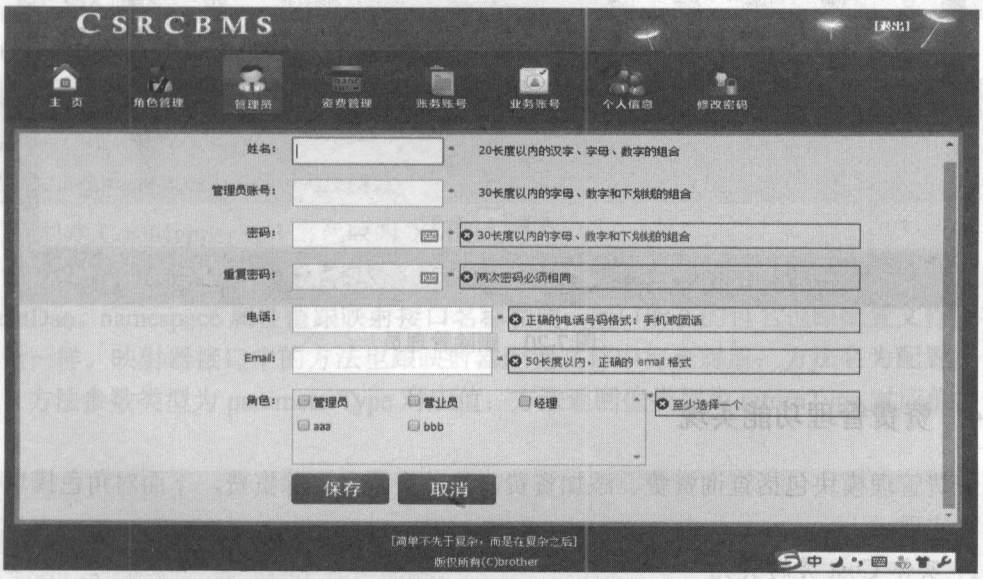

图 7-18　添加管理员

(3) 修改管理员的用户界面如图 7-19 所示。

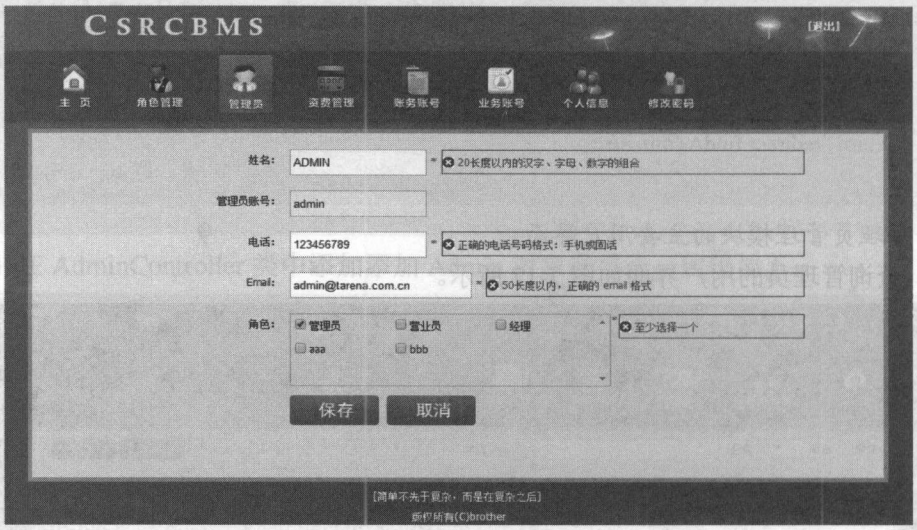

图 7-19 修改管理员

(4) 删除管理员的用户界面如图 7-20 所示。

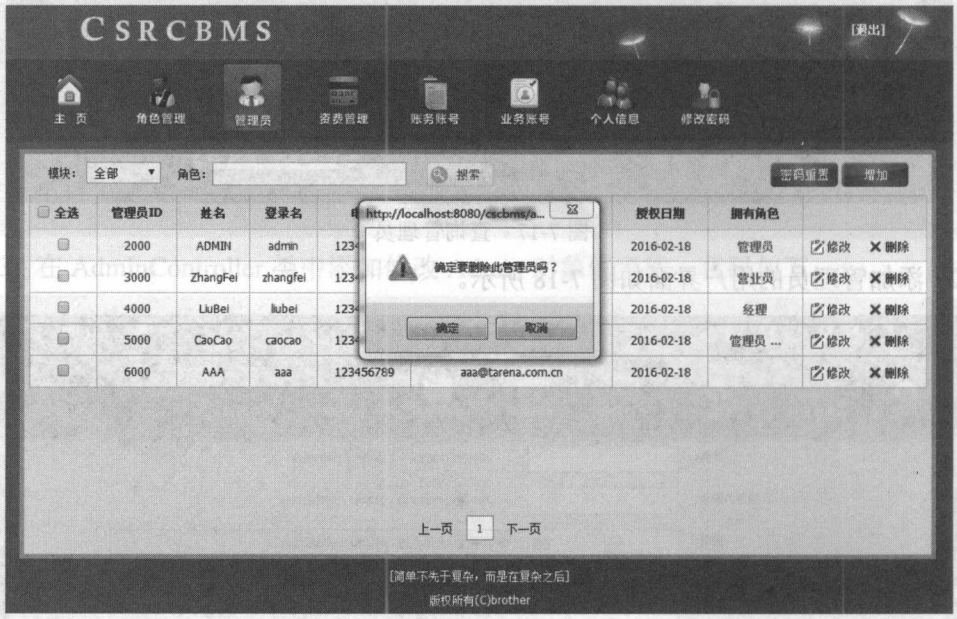

图 7-20 删除管理员

7.4.4 资费管理功能实现

资费管理模块包括查询资费、添加资费、修改资费和删除资费，下面对角色模块的功能进行实现。

1) 实体 Cost 类的实现

通常实体类的成员变量和数据库资费表的字段一一对应，且成员变量名和表的字段名

应尽量一致，实体 Cost 与数据库表 cost 对应，基于这种约定，Cost 类代码实现如下：

```
public class Cost {
    private Integer costId;//资费编号
    private String name;//资费名称
    private Integer baseDuration;// 基本时长
    private Double baseCost;// 基本费用
    private Double unitCost;// 单位费用
    private String status;//资费状态
    private String descr;//资费说明
    private Timestamp creatTime;//资费创建时间
    private Timestamp startTime;//资费启用时间
    private String costType;//资费类型
//属性对应的getter和setter方法略
}
```

2) 创建映射接口 CostDAO

该接口定义对数据库增加资费、删除资费、修改资费和查询资费方法。

(1) CostDao 提供多种查找资费方法，以便系统扩展。代码如下：

```
@MyBatisRepository
public interface CostDao {
List<Cost> findAll();//查询所有资费对象
List<Cost> findByPage(Page page);//分页查询资费对象
int findRows();//查询资费记录的总数
Cost findById(int id);//根据资费编码返回资费对象
}
```

(2) CostDao 接口提供新增、修改和删除资费的方法，代码如下：

```
void save(Cost cost);//保存资费对象
void update(Cost cost); //资费角色
void delete(int id);//删除资费对象根据资费Id
```

3) 创建 CostMapper.xml 角色映射 XML 文件

定义 SQL 语句与 CostDao 映射接口方法相匹配，设置 mapper 元素的 namespace 属性为 CostDao。namespace 属性值跟映射接口名称相同，接口所在的包名也跟配置文件所在包名完全一样。映射器接口中的方法也跟映射器配置文件中完全对应：方法名为配置文件中 id 值，方法参数类型为 parameterType 对应值；方法返回值类型为 returnType 对应值。代码如下：

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//ibatis.apache.org//DTD Mapper 3.0//EN"
 "http://ibatis.apache.org/dtd/ibatis-3-mapper.dtd">
```

```
<mapper namespace = "com.zds.dao.CostDao">
</mapper>
```

(1) 在 CostMapper.xml 接口添加查询语句。代码如下：

```
<!--查询所有资费记录并资费编号排序 -->
<select id = "findAll" resultType = "com.cscbms.entity.Cost">
    select * from cost order by cost_id
</select>
<!--查询所有资费记录并资费编号排序 -->
<select id = "findById" parameterType = "int"
        resultType = "com.cscbms.entity.Cost">
        select * from cost where cost_id = #{id}
    </select>
<!—分页查询资费记录并按资费编号排序 -->
<select id = "findByPage"
        parameterType = "com.cscbms.entity.page.Page"
        resultType = "com.cscbms.entity.Cost">
        select * from cost order by cost_id limit #{begin}, #{end}
    </select>
<!—查询所有资费记录数-->
<select id = "findRows" resultType = "int">
        select count(*) from cost
    </select>
```

在条件查询设置字段值时，用#{参数名}获取参数值，如果参数名为 id，可以写成#{id}；如果参数是 Page 类 page 对象，Page 类有 begin、end 属性，可以用#{begin}、#{end}获取 page 的 begin、end 的值。id 为“findRows”的查询语句查询所有资费记录数。

(2) 为了对资费对象进行新增、修改和删除操作，在 CostMapper.xml 中定义相应 SQL 语句，代码如下：

```
//添加资费记录
<insert id = "save" parameterType = "com.cscbms.entity.Cost">
    <![CDATA[
        insert into cost(name, base_duration, base_cost, unit_cost, status, descry, creatime, startime, cost_type)
values(
            #{name, jdbcType = VARCHAR},
            #{baseDuration, jdbcType = INTEGER},
            #{baseCost, jdbcType = DOUBLE},
            #{unitCost, jdbcType = DOUBLE},
            #{status, jdbcType = CHAR},
            #{descr, jdbcType = VARCHAR},
```

```
            #{creatTime, jdbcType = TIMESTAMP},
            #{startTime, jdbcType = TIMESTAMP},
            #{costType, jdbcType = CHAR}
        )
    ]]>
    </insert>
//修改资费记录
<update id = "update" parameterType = "com.cscbms.entity.Cost">
        <![CDATA[
            update cost set
            name = #{name},
            base_duration = #{baseDuration, jdbcType = NUMERIC},
            base_cost = #{baseCost, jdbcType = NUMERIC},
            unit_cost = #{unitCost, jdbcType = NUMERIC},
            descr = #{descr, jdbcType = VARCHAR},
            cost_type = #{costType, jdbcType = CHAR}
            where cost_id = #{costId}
        ]]>
    </update>
//删除资费记录
<delete id = "delete" parameterType = "int">
        delete from cost where cost_id = #{id}
    </delete>
```

4) 创建 CostCotroller 类

在类级别上添加@Controller 注解，当 Spring 自动扫描到类中@Controller 注解时，在 Spring 容器中创建 CostController 对象。在类级别上添加@RequestMapping 注解，设置属性 value 的值为“/cost”，SpringMVC 框架将请求路径中包含“/cost”的请求映射到 CostController 对象，并由 CostController 对象处理该请求；在类级别上@SessionAttributes，声明 Session 属性；定义成员变量 CostDao 对象，在 CostDao 对象上添加@Resource，SpringMVC 会自动注入 CostDao 对象，利用 CostDao 对象的接口方法对资费对象实现持久化操作。代码如下：

```
@Controller
@RequestMapping("/cost")
@SessionAttributes("costPage")
public class CostController {
    @Resource
    private CostDao costDao;
}
```

(1) 在 CostController 类中添加保存 Cost 对象的方法，代码如下：

```
@RequestMapping("/addCost.do")
    public String add(Cost cost) {
        cost.setStatus("1");
        cost.setCreatTime(
            new Timestamp(System.currentTimeMillis()));
        costDao.save(cost);
        return "redirect:findCost.do";
    }
```

在上面代码中@RequestMapping 注解设置请求映射，当请求/addCost.do 的时候调用add()方法处理请求。

(2) 在 CostController 类中添加查询 Cost 对象的方法，代码如下：

```
@RequestMapping("/findCost.do")
    public String find(CostPage page, Model model) {
        page.setRows(costDao.findRows());//设置查询记录总条数
        model.addAttribute("costPage", page);// 保存page对象到model属性中
        List<Cost> list = costDao.findByPage(page);//分页查询
        model.addAttribute("costs", list);//保存资费列表到model属性中
        return "cost/costList";
    }
```

find()方法有 CostPage 和 Model 类型的两个参数，CostPage 对象封装分页查询的数据，将分页查询的结果保存在 model 属性中，以便在前台页展示资费列表。

5) 资费管理模块的主要用户界面

(1) 查询资费的用户界面如图 7-21 所示。

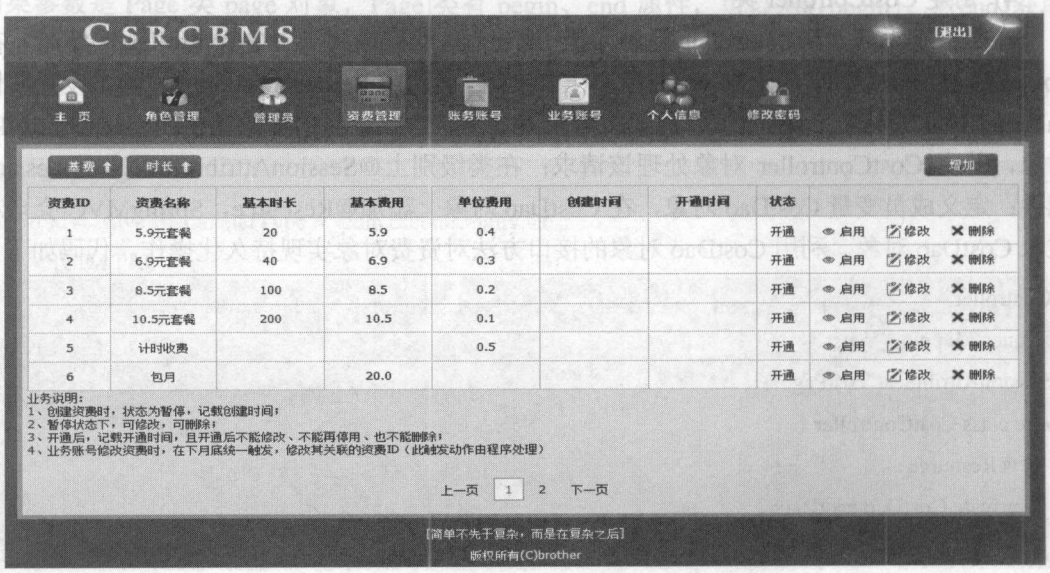

图 7-21 查询资费

(2) 添加资费的用户界面如图 7-22 所示。

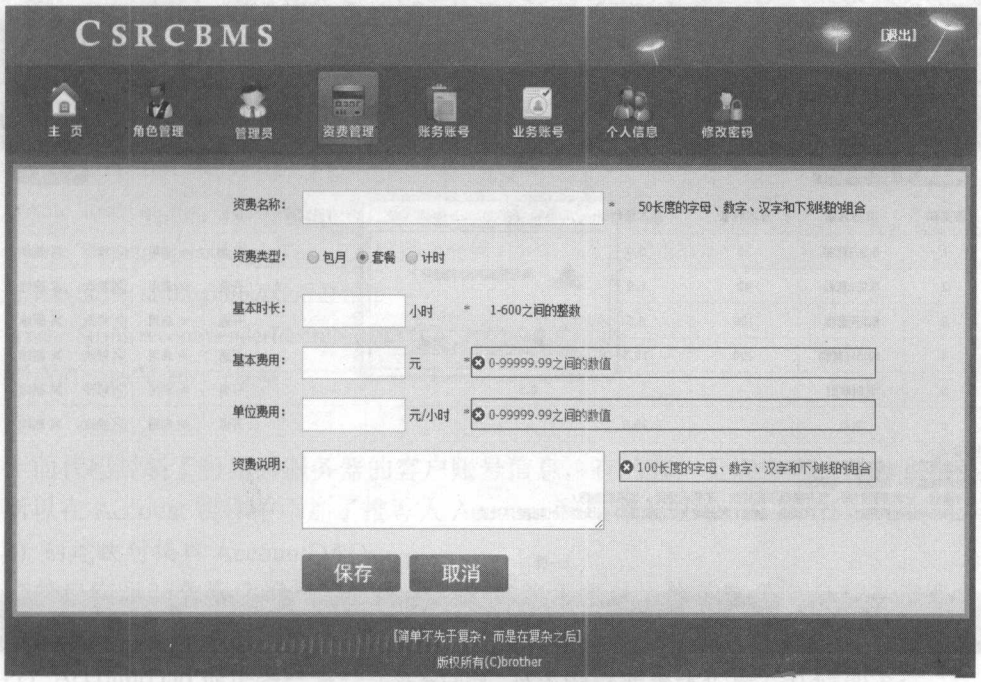

图 7-22 添加资费

(3) 修改资费的用户界面如图 7-23 所示。

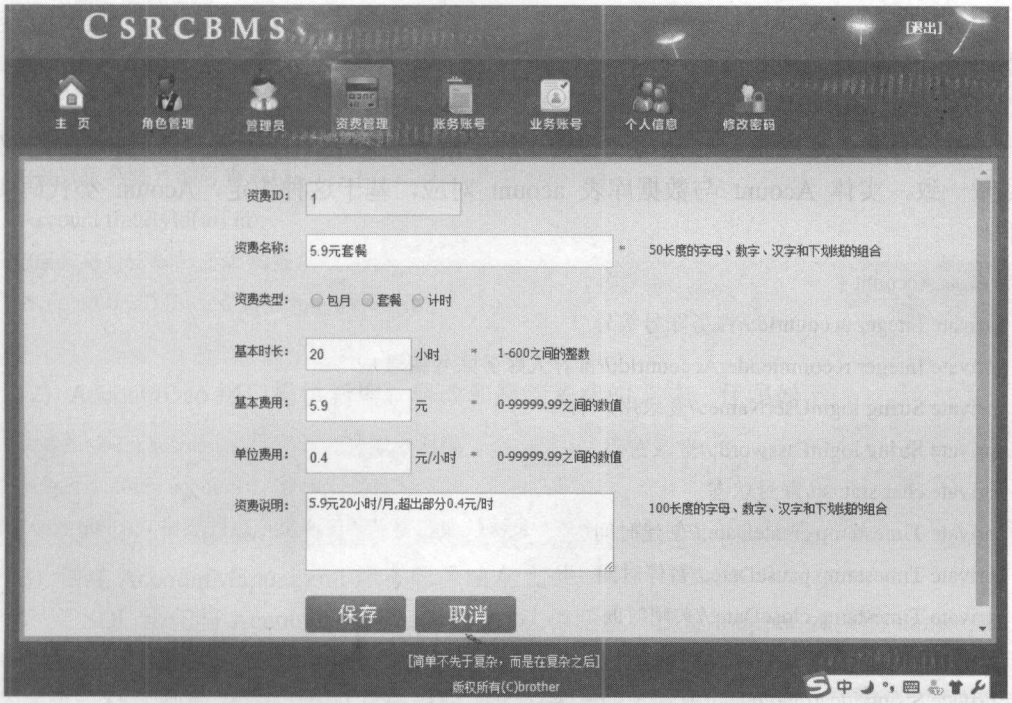

图 7-23 修改资费

(4) 删除资费的用户界面如图 7-24 所示。

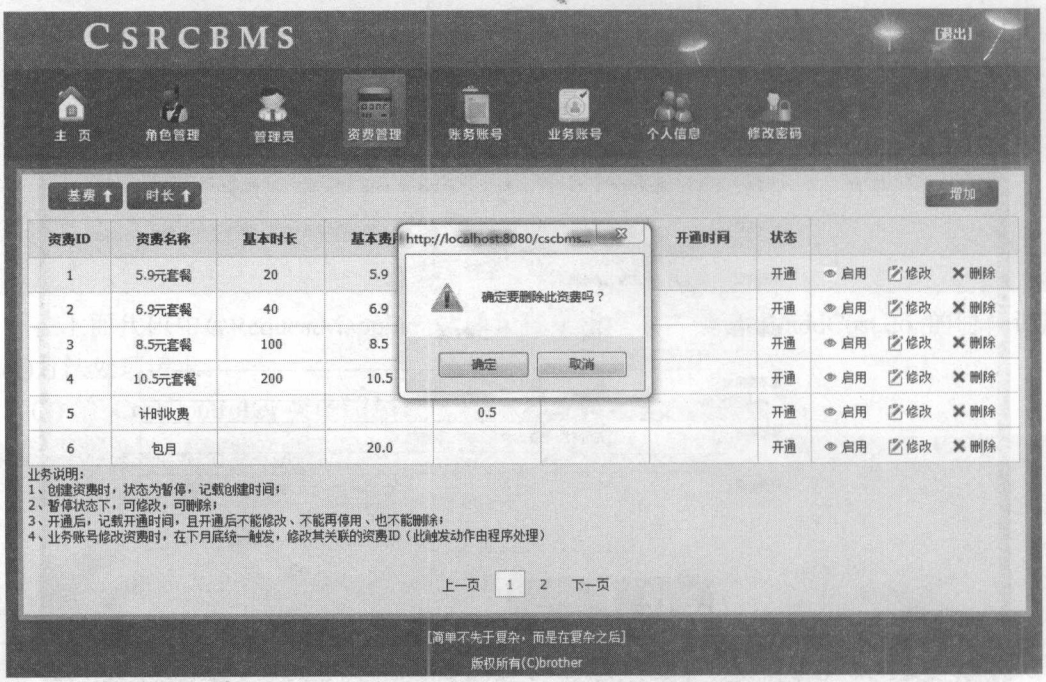

图 7-24 删除资费

7.4.5 账务账号管理功能实现

账务账号管理模块包括查询账务账号、添加账务账号、修改账务账号，下面对账务账号模块的功能进行实现。

1) 实体 Account 类的实现

通常实体类的成员变量和数据库资费表的字段一一对应，且成员变量名和表的字段名应尽量一致，实体 Acount 与数据库表 acount 对应，基于这种约定，Account 类代码实现如下：

```
public class Account {
    private Integer accountId;//账务账号编号
    private Integer recommenderAccountId;//推荐人账务账号编号
    private String loginUserName;//登录用户名
    private String loginPassword;//登录密码
    private char status;//账号状态
    private Timestamp createDate;//创建时间
    private Timestamp pauseDate;//暂停时间
    private Timestamp closeDate;//关闭时间
    private String realName;//真实姓名
    private String idcardNo;//身份证号
    private Date birthDate;//出生日期
```

```
    private String gender;//性别
    private String occupation;//职业
    private String telephone;//电话号码
    private String email;//邮箱地址
    private String mailAddress;//邮寄地址
    private String zipCode;//邮编号码
    private String qq;//qq号码
    private Timestamp lastLoginTime;//最后登录时间
    private String lastLoginIp;//最后登录Ip
    private String recommenderIdcardNo;//推荐人身份证编号
//属性对应的getter和setter方法略
}
```

上面代码封装了租用云服务器的客户账号信息，有的客户是通过客户推荐租用该服务的，所以在 Account 实体中添加了推荐人 Account 和推荐人 IDCard。

2) 创建映射接口 AccountDAO

该接口定义对数据库增加账务账号、删除账务账号、修改账务账号和查询账务账号方法。

(1) AccountDao 提供多种查找账务账号的方法，以便系统扩展。代码如下：

```
@MyBatisRepository
public interface AccountDao {
//分页查询账务账号记录
    List<Account> findByPage(Page page);
    //查询账务账号总记录数
    int findRows(Page page);
    //根据账务账号编号查询Id
    Account findById(int id);
    //根据身份证编号查询账务账号
    Account findByIdcardNo(String idcardNo);
    }
```

(2) AccountDao 接口提供新增、修改和删除资费的方法，代码如下：

```
void save(Account account);//保存找账务账号对象
void update(Account account); //修改账务账号对象
void delete(int id);// 根据找账务账号id删除账务账号对象
```

3) 创建 AccountMapper.xml 账务账号映射文件

定义 SQL 语句与 AccountDao 映射接口方法相匹配，设置 mapper 元素的 namespace 属性为 Account。namespace 属性值跟映射接口名称相同，接口所在的包名也跟配置文件所在包名完全一样。映射器接口中的方法也跟映射器配置文件中完全对应：方法名为配置文件中 id 值，方法参数类型为 parameterType 对应值；方法返回值类型为 returnType 对应值。代

码如下：

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//ibatis.apache.org//DTD Mapper 3.0//EN"
 "http://ibatis.apache.org/dtd/ibatis-3-mapper.dtd">
<mapper namespace = "com.zds.dao.AccountDao">
</mapper>
```

(1) 在 AccountMapper.xml 接口添加查询语句，分别是分页查询、返回查询记录数、根据账务账号编号查询和根据身份证编号查询。代码如下：

```
<select id = "findByPage"
            parameterType = "com.cscbms.entity.page.Page"
            resultType = "com.cscbms.entity.Account">
                    select * from account
          <!—定义查询条件-->
<where>
            <if test = "idcardNo != null &amp;&amp; idcardNo.length()>0">
                and idcard_no = #{idcardNo}
            </if>
            <if test = "realName != null &amp;&amp; realName.length()>0">
                and real_name = #{realName}
            </if>
<if test = "loginUserName != null &amp;&amp;
      loginUserName.length() > 0">
                and login_name = #{loginUserName}
            </if>
            <if test = 'status != null &amp;&amp; !status.equals("-1")'>
                and status = #{status}
            </if>
            </where>
            order by account_id limit #{begin}, #{end}
      </select>
<select id = "findRows"
            parameterType = "com.cscbms.entity.page.Page"
            resultType = "int">
            select count(*) from account
            <where>
                <if test = "idcardNo != null &amp; &amp; idcardNo.length()>0">
                    and idcard_no = #{idcardNo}
                </if>
```

```
            <if test = "realName != null &amp;&amp; realName.length()>0">
                and real_name = #{realName}
            </if>
            <if test = "loginUserName != null &amp;&amp;
            loginUserName.length()>0" >
            and login_name = #{loginName}
            </if>
            <if test = 'status != null &amp;&amp; !status.equals("-1")'>
                and status = #{status}
            </if>
        </where>
    </select>
<select id = "findById"
        parameterType = "int"
        resultType = "com.cscbms.entity.Account">
        select a.*, r.idcard_no from account a
        left join account r on a.recommenderId = r.accountid
        where a.account_id = #{id}
    </select>
    <select id = "findByidcardNo"
        parameterType = "string"
        resultType = "com.cscbms.entity.Account">
        select * from account where idcard_no = #{idcardNo}
    </select>
    </select>
```

在条件查询设置字段值时，用#{参数名}获取参数值，如果参数名为 id，可以写成#{id}；如果参数是 Page 类 page 对象，Page 类有 begin、end 属性，可以用 #{begin}、#{end}获取 page 的 begin、end 的值。id 为“findRows”的条件查询所有账务账号记录数。

(2) 为了对账务账号对象进行新增、修改操作，在 AccountMapper.xml 中定义相应 SQL 语句，代码如下：

```
//添加账务账号记录
<insert id = "save"
        parameterType = "com.cscbms.entity.Account">
    <![CDATA[
        insert into account(recommender_id, login_name, login_password,
        status, create_date, pause_date, close_date, real_name, idcard_no,
        birthDate, gender, occupation, telephone, email, mailAddress, zipCode, qq,
        last_login_time, last_login_ip) values(
```

```
            #{recommenderAccountId, jdbcType = NUMERIC},
            #{loginUserName, jdbcType = VARCHAR},
            #{loginPassword, jdbcType = VARCHAR},
            #{status, jdbcType = CHAR},
            #{createDate, jdbcType = TIMESTAMP},
            #{pauseDate, jdbcType = TIMESTAMP},
            #{closeDate, jdbcType = TIMESTAMP},
            #{realName, jdbcType = VARCHAR},
            #{idcardNo, jdbcType = VARCHAR},
            #{birthDate, jdbcType = DATE},
            #{gender, jdbcType = CHAR},
            #{occupation, jdbcType = VARCHAR},
            #{telephone, jdbcType = VARCHAR},
            #{email, jdbcType = VARCHAR},
            #{mailAddress, jdbcType = VARCHAR},
            #{zipCode, jdbcType = CHAR},
            #{qq, jdbcType = VARCHAR},
            #{lastLoginTime, jdbcType = TIMESTAMP},
            #{lastLoginIp, jdbcType = VARCHAR}
        )
    ]]>
    </insert>
//修改账务账号记录
<update id = "update"
        parameterType = "com.cscbms.entity.Account">
    <![CDATA[
        update account set
            recommender_id = #{recommenderAccountId },
            real_name = #{realName },
            telephone = #{telephone },
            email = #{email },
            occupation = #{occupation },
            gender = #{gender },
            mailaddress = #{mailAddress },
            zipcode = #{zipCode },
            qq = #{qq }
        where account_id = #{accountId}
    ]]>
    </update>
```

4) 创建 AccountCotroller 类

在类级别上添加@Controller 注解，当 Spring 自动扫描到类中@Controller 注解时，在 Spring 容器中创建 AccountController 对象。在 AccountCotroller 类级别上添加@RequestMapping 注解，设置属性 value 的值为“/Account”。SpringMVC 框架将请求路径中包含“/Account”的请求映射到 AccountController 对象，并由 AccountController 对象处理该请求；在 AccountCotroller 类级别上@SessionAttributes，声明 Session 属性；定义成员变量 AccountDao 对象，在 AccountDao 对象上添加@Resource，SpringMVC 会自动注入 AccountDao 对象，利用 AccountDao 对象的接口方法对 Account 对象实现持久化操作。代码如下：

```
@Controller
@RequestMapping("/account")
@SessionAttributes("accountPage")
public class AccountController extends BaseController {
    @Resource
    private AccountDao accountDao;
}
```

(1) 在 AccountController 类中添加查询 Account 对象的方法，代码如下：

```
@RequestMapping("/findAccount.do")
    public String find(AccountPage page, Model model) {
        page.setRows(accountDao.findRows(page));
        model.addAttribute("accountPage", page);
        List<Account> list = accountDao.findByPage(page);
        model.addAttribute("accounts", list);
        return "account/accountList";
}
```

find()方法有 AccountPage 和 Model 类型的两个参数，AccountPage 对象封装分页查询的数据，将分页查询的结果保存在 model 属性中，以便在前台页展示账务账号列表。

(2) 在 AccountController 类中添加保存 Account 对象的方法，代码如下：

```
@RequestMapping("/addAccount.do")
    public String add(Account account) {
        account.setStatus('0');
        account.setCreateDate(
            new Timestamp(System.currentTimeMillis()));
        accountDao.save(account);
        return "redirect:findAccount.do";
    }
```

在上面代码中@RequestMapping 注解设置请求映射，当请求/addAccount.do 的时候调用 add()方法处理请求，新增账务账号默认的状态为开通，状态为'0'表示开通状态。

(3) 在 AccountController 类中添加修改 Account 对象的方法，代码如下：

```
@RequestMapping("/updateAccount.do")
    public String update(Account account) {
        accountDao.update(account);
        return "redirect:findAccount.do";
    }
```

5) 账务账号管理模块的主要用户界面

(1) 查询账务账号的用户界面如图 7-25 所示。

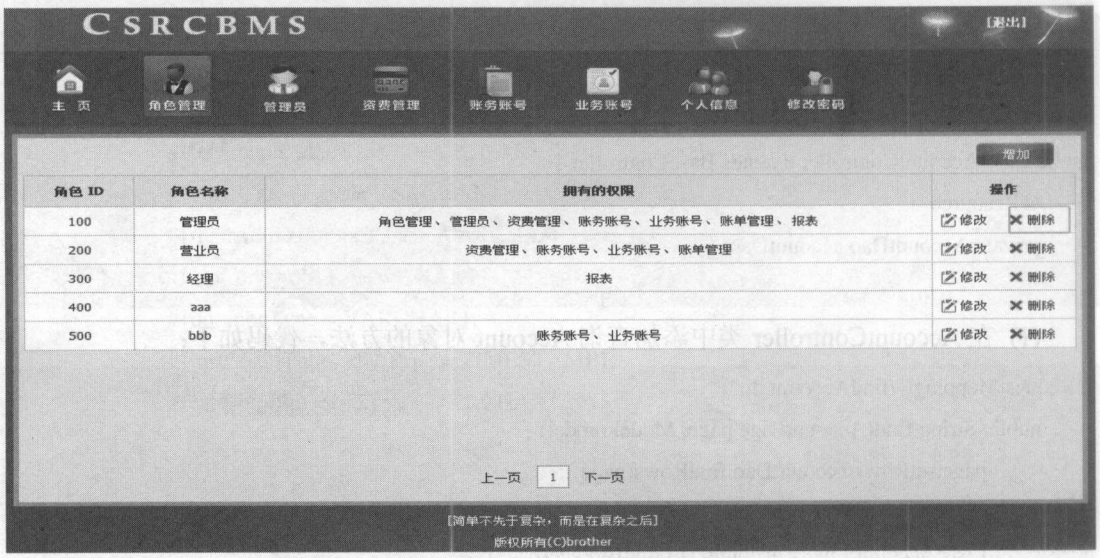

图 7-25　查询账务账号

(2) 添加账务账号的用户界面如图 7-26 所示。

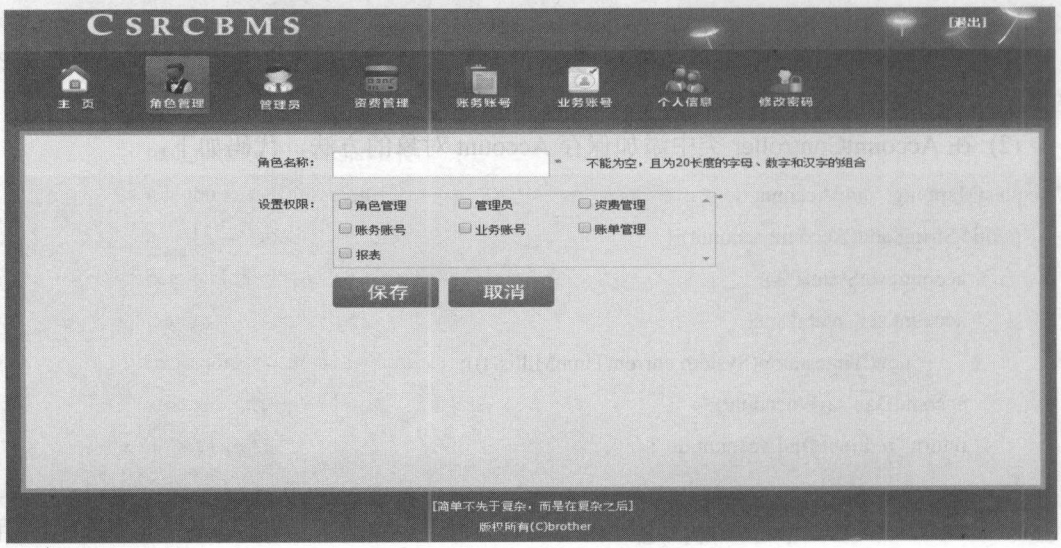

图 7-26　添加账务账号

(3) 修改账务账号的用户界面如图 7-27 所示。

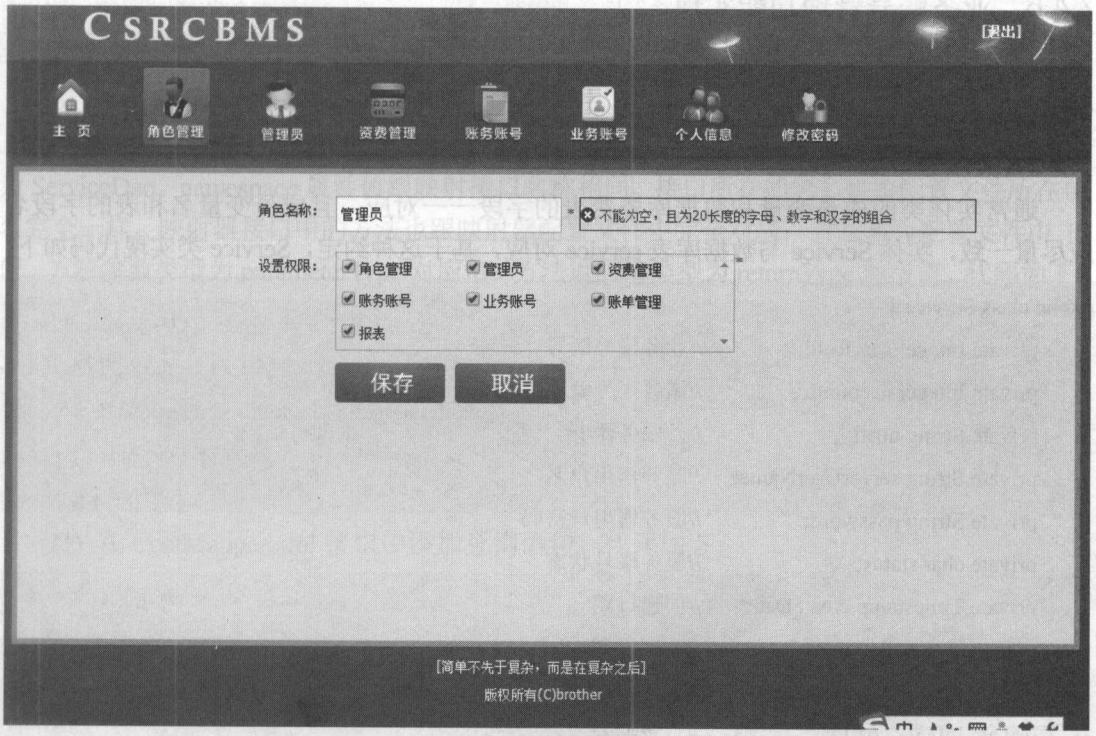

图 7-27　修改账务账号

(4) 删除账务账号的用户界面如图 7-28 所示。

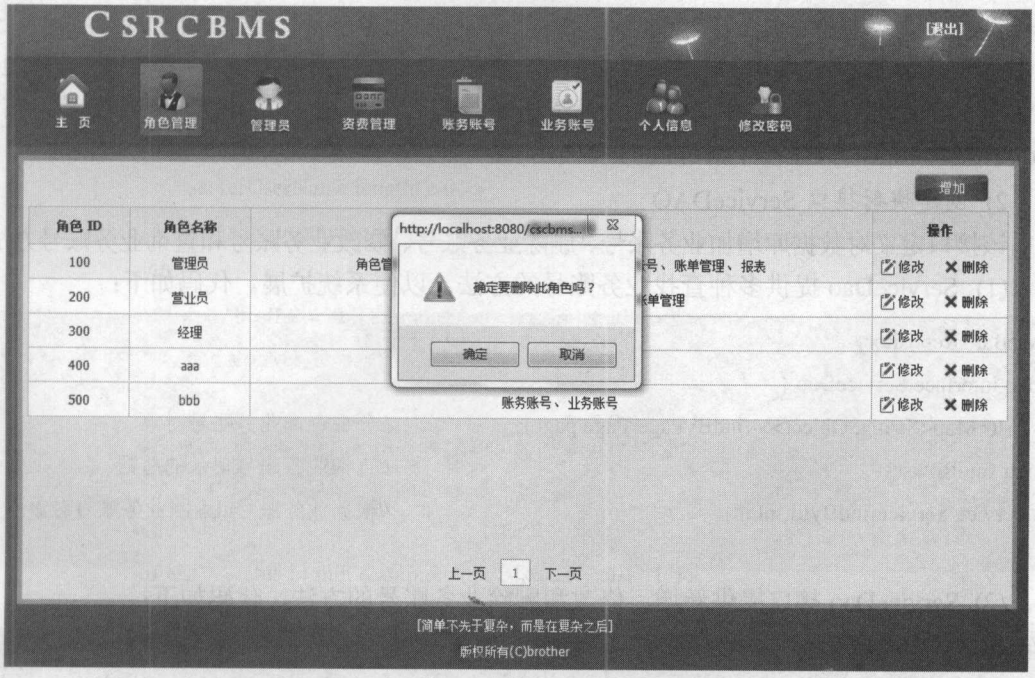

图 7-28　删除账务账号

7.4.6 业务账号管理功能实现

业务账号管理模块包括查询业务账号、添加业务账号、修改业务账号和删除业务账号，下面对业务账号模块的功能进行实现。

1) 实体 Service 类的实现

通常实体类的成员变量和数据库资费表的字段一一对应，且成员变量名和表的字段名应尽量一致，实体 Service 与数据库表 service 对应，基于这种约定，Service 类实现代码如下：

```
public class Service {
    private Integer serviceId;          //业务账号编号
    private Integer accountId;          //账务账号编号
    private String hostIp;              //云服务器 Ip
    private String serverUserName;      //服务器用户名
    private String password;            //服务器用户密码
    private char status;                //服务账号状态
    private Timestamp createDate;       //创建日期
    private Timestamp pauseDate;        //暂停日期
    private Timestamp closeDate;        //关闭日期
    private Integer costId;             //资费编号
    private String idcardNo;            //身份证编号
    //setter和getter方法略
}
```

在 Service 类的实现中，考虑到添加一个业务账号，通常是已有的一个账务账号开通一个业务，一个业务关联一种资费，所以添加账务账号编号和资费编号的成员变量，实现 Serice、Account、Cost 三个实体间的关联。

2) 创建映射接口 ServiceDAO

该接口定义对数据库增加业务账号、删除业务账号、修改业务账号和查询业务账号方法。

(1) ServiceDao 提供多种查找业务账号的方法，以便系统扩展。代码如下：

```
@MyBatisRepository
public interface ServiceDao {
    List<Map<String, Object>> findByPage(Page page);        //分页查询业务账号对象
    int findRows();                                         //查询业务账号记录的总数
    Service ServicefindById(int id);                        //根据业务账号id返回业务账号对象
}
```

(2) ServiceDao 接口提供新增、修改和删除业务账号的方法，代码如下：

```
void save(Service service);          //保存业务账号
void update(Service service);        //修改业务账号
```

```
void deleteByAccount(int accountId);          //根据账务账号 Id 删除业务账号
void updateStatus(Service service);           //修改业务账号的状态
void pauseByAccount(int accountId);           //根据账务账号 Id 暂停业务账号
```

3) 创建 ServiceMapper.xml 业务账号映射文件

定义 SQL 语句与 ServiceDao 映射接口方法相对应，设置 mapper 元素的 namespace 属性为 ServiceDao。namespace 属性值跟映射接口名称相同，接口所在的包名也跟配置文件所在包名完全一样。映射器接口中的方法也跟映射器配置文件中完全对应：方法名为配置文件中 id 值，方法参数类型为 parameterType 对应值；方法返回值类型为 returnType 对应值。代码如下：

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//ibatis.apache.org//DTD Mapper 3.0//EN"
 "http://ibatis.apache.org/dtd/ibatis-3-mapper.dtd">
<mapper namespace = "com.zds.dao.ServiceDao">
</mapper>
```

(1) 在 CostMapper.xml 接口中添加查询语句。代码如下：

```
<!—分页查询Service -->
<select id = "findByPage"
        parameterType = "com.cscbms.entity.page.Page"
        resultType = "com.cscbms.vo.ServiceVo">
        select   s.*, a.real_name, a.idcard_no, c.name costName, c.descr
        from service s
        inner join account a on a.account_id = s.account_id
        inner join cost c on c.cost_id = s.cost_id
        <where>
            <if test = "serverUserName != null &amp;&amp;
                serverUserName.length()>0" >
                and s.os_username = #{serverUserName}
            </if>
            <if test = "hostIp != null &amp;&amp; hostIp.length()>0">
                and s.server_host = #{hostIp}
            </if>
            <if test = "idcardNo != null &amp;&amp; idcardNo.length()>0">
                and a.idcard_no = #{idcardNo}
            </if>
            <if test = 'status != null &amp;&amp; !status.equals("-1")'>
                and s.status = #{status}
            </if>
        </where>
```

```
                order by s.service_id limit #{begin}, #{end}
    </select>
<!—条件查询返回满足添加的记录数 -->
<select id = "findRows"
        parameterType = "com.cscbms.entity.page.Page"
        resultType = "int">
        select
            count(*)
        from service s
        inner join account a on a.account_id = s.account_id
        inner join cost c on c.cost_id = s.cost_id
        <where>
            <if test = "serverUserName != null &amp;&amp;
                serverUserName.length()>0" >
                and s.os_username = #{serverUserName}
            </if>
            <if test = "hostIp != null &amp;&amp; hostIp.length()>0">
                and s.server_host = #{hostIp}
            </if>
            <if test = "idcardNo != null &amp;&amp; idcardNo.length()>0" >
                and a.idcard_no = #{idcardNo}
            </if>
            <if test = 'status != null &amp; &amp; !status.equals("-1")' >
                and s.status = #{status}
            </if>
        </where>
    </select>
<!—按业务编号查询-->
<select id = "findById"
        parameterType = "int"
        resultType = "com.cscbms.entity.Service">
        select * from service where service_id = #{id}
</select>
```

在条件查询设置字段值时，用#{参数名}获取参数值，如果参数名为 id，可以写成 #{id}；如果参数是 Page 类 page 对象，Page 类有 begin、end 属性，可以用 #{begin}、#{end}获取 page 的 begin、end 的值。id 为“findRows”的查询语句查询所有业务账号记录数。

(2) 为了对业务账号对象进行新增、修改和删除操作，在 ServiceMapper.xml 中定义相应 SQL 语句，代码如下：

```
//添加业务账号记录
<insert id = "save" parameterType = "com.cscbms.entity.Service">
        insert into service(account_id, server_host, os_username, password, create_date, create_date,
close_date,  cost_id) values(
            values (#{accountId},
            #{hostIp },
            #{serverUserName },
            #{password },
            #{status },
            #{createDate },
            #{pauseDate },
            #{closeDate },
            #{costId }
        )
    </insert>
//修改业务账号记录，只允许修改资费
<update id = "update" parameterType = "com.cscbms.entity.Service">
        update service set cost_id = #{costId}
        where service_id = #{serviceId}
    </update>
//根据账务账号删除业务账号记录
<update id = "deleteByAccount" parameterType = "int">
        update service set status = '2', closedate = NOW()
        where account_id = #{accountId}
    </update>
```

4) 创建 ServiceCotroller 类

在类级别上添加@Controller 注解，当 Spring 自动扫描到类中@Controller 注解时，在 Spring 容器中创建 ServiceController 对象。在类级别上添加@RequestMapping 注解，设置属性 value 的值为“/service”，SpringMVC 框架将请求路径中包含“/service”的请求映射到 ServiceController 对象，并由 ServiceController 对象处理该请求；在 ServiceController 类级别上添加@SessionAttributes，声明 Session 属性。代码如下：

```
@Controller
@RequestMapping("/service")
@SessionAttributes("servicePage")
public class ServiceController extends BaseController {
    @Resource
    private ServiceDao serviceDao;
```

```
    @Resource
    private AccountDao accountDao;
    @Resource
    private CostDao costDao;
```

在 ServiceController 类中添加了 ServiceDao、AccountDao、CostDao 类型的成员变量，并添加@Resource，Spring 容器自动注入 bean 对象初始化这些成员变量，利用 Dao 层提供的接口对象数据库操作。

(1) 在 ServiceController 类中添加查询 Service 对象的方法，代码如下：

```
@RequestMapping("/findService.do")
    public String find(ServicePage page, Model model) {
        page.setRows(serviceDao.findRows(page));
        model.addAttribute("servicePage", page);
        //分页查找业务对象列表
        List<Map<String, Object>> list =
                serviceDao.findByPage(page);
        model.addAttribute("services", list);
        //跳转到展示业务账号列表页面
        return "service/serviceList";
    }
```

find()方法有 ServicePage 和 Model 类型的两个参数，ServicePage 对象封装分页查询的数据，将分页查询的结果保存在 model 属性中，以便在前台页展示业务账号列表。

(2) 在 ServiceController 类中添加保存 Service 对象的方法，代码如下：

```
@RequestMapping("/addService.do")
    public String add(Service service) {
        //设置业务账号为开通状态
        service.setStatus('0');
        //设置业务账号创建时间
        service.setCreateDate(
                new Timestamp(System.currentTimeMillis()));
        serviceDao.save(service);
        return "redirect:findService.do";
    }
```

在上面代码中@RequestMapping 注解设置请求映射，当请求/addService.do 的时候调用 add()方法处理请求，处理完成后将请求转发到 findService.do 重新查询业务账号表，使添加记录在前台页面同步更新。

5) 业务账号管理模块的主要用户界面

(1) 查询业务账号的用户界面如图 7-29 所示。

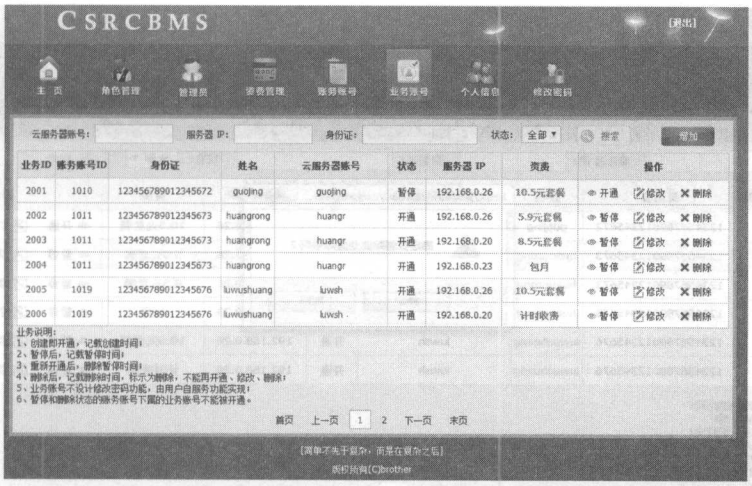

图 7-29 查询业务账号

(2) 添加业务账号的用户界面如图 7-30 所示。

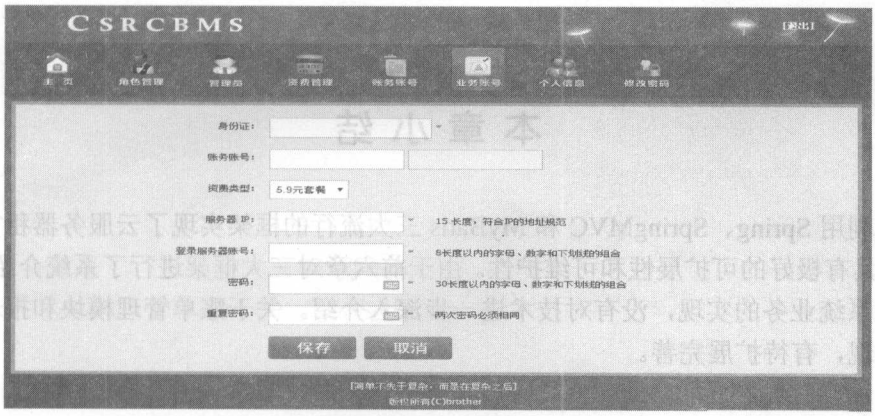

图 7-30 添加业务账号

(3) 修改业务账号的用户界面如图 7-31 所示。

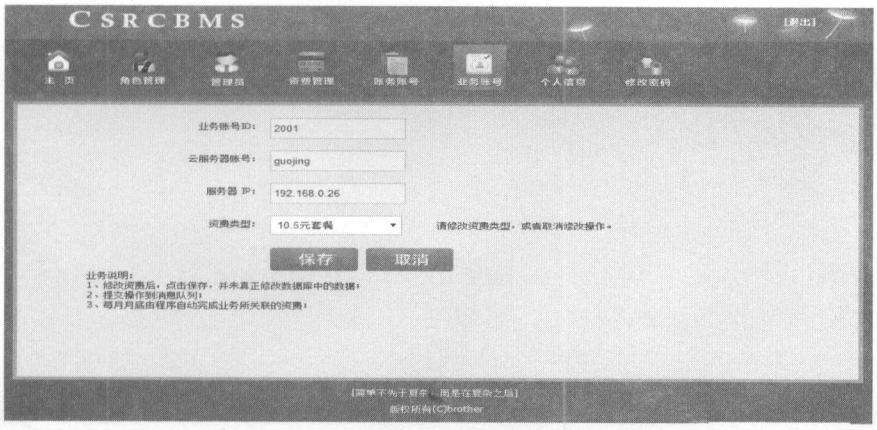

图 7-31 修改业务账号

(4) 删除业务账号的用户界面如图 7-32 所示。

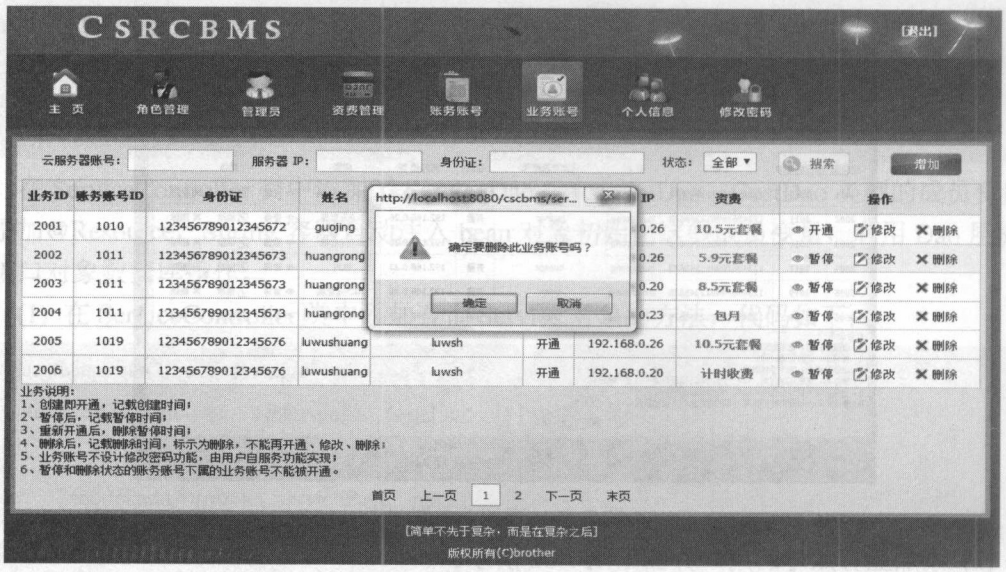

图 7-32　删除业务账号

本章小结

本章利用 Spring、SpringMVC 和 MyBatis 三大流行的框架实现了云服务器租赁资费管理系统，具有极好的可扩展性和可维护性。由于前六章对三大框架进行了系统介绍和学习，本章重在系统业务的实现，没有对技术进一步深入介绍。关于账单管理模块和报表模块本章没有实现，有待扩展完善。

XDUP 488100
封面设计：佳易传播 WWW.SXJYCB.COM

Java EE框架技术

本书对当前企业使用较多的、流行的三大技术框架SpringMVC、Spring和MyBatis的基本知识和使用方法进行了详细的讲解。全书共七章。第一章主要介绍MyBatis开发入门知识；第二章主要介绍MyBatis配置选项；第三章主要介绍MyBatis映射器(Mapper)；第四章主要介绍Spring核心技术；第五章主要介绍SpringMVC；第六章主要介绍SpringMVC、Spring、MyBatis三个框架的集成；第七章主要是项目实战部分。本书在讲解知识点的同时还提供了丰富的案例，每章末均给出一定量的练习题，以帮助学生巩固学习效果，加深对相关知识点的理解。

ISBN 978-7-5606-4589-6
9 787560 645896 >
定价：26.00元